

Learning Ada

The complete contents
of learn.adacore.com

LEARN.
ADACORE.COM

Learning Ada

Release 2025-12

Various authors

Dec 27, 2025

CONTENTS

I	Introduction to Ada	3
1	Introduction	7
1.1	History	7
1.2	Ada today	7
1.3	Philosophy	8
1.4	SPARK	8
2	Imperative language	9
2.1	Hello world	9
2.2	Imperative language - If/Then/Else	10
2.3	Imperative language - Loops	13
2.3.1	For loops	13
2.3.2	Bare loops	14
2.3.3	While loops	15
2.4	Imperative language - Case statement	16
2.5	Imperative language - Declarative regions	18
2.6	Imperative language - conditional expressions	20
2.6.1	If expressions	20
2.6.2	Case expressions	21
3	Subprograms	23
3.1	Subprograms	23
3.1.1	Subprogram calls	24
3.1.2	Nested subprograms	25
3.1.3	Function calls	26
3.2	Parameter modes	28
3.3	Subprogram calls	28
3.3.1	In parameters	28
3.3.2	In out parameters	29
3.3.3	Out parameters	30
3.3.4	Forward declaration of subprograms	31
3.4	Renaming	31
4	Modular programming	35
4.1	Packages	35
4.2	Using a package	37
4.3	Package body	37
4.4	Child packages	39
4.4.1	Child of a child package	41
4.4.2	Multiple children	42
4.4.3	Visibility	43
4.5	Renaming	45
5	Strongly typed language	47
5.1	What is a type?	47

5.2	Integers	47
5.2.1	Operational semantics	49
5.3	Unsigned types	50
5.4	Enumerations	51
5.5	Floating-point types	52
5.5.1	Basic properties	52
5.5.2	Precision of floating-point types	53
5.5.3	Range of floating-point types	54
5.6	Strong typing	56
5.7	Derived types	58
5.8	Subtypes	60
5.8.1	Subtypes as type aliases	62
6	Records	65
6.1	Record type declaration	65
6.2	Aggregates	66
6.3	Component selection	66
6.4	Renaming	67
7	Arrays	71
7.1	Array type declaration	71
7.2	Indexing	74
7.3	Simpler array declarations	75
7.4	Range attribute	76
7.5	Unconstrained arrays	78
7.6	Predefined array type: String	79
7.7	Restrictions	81
7.8	Returning unconstrained arrays	82
7.9	Declaring arrays (2)	83
7.10	Array slices	84
7.11	Renaming	85
8	More about types	89
8.1	Aggregates: A primer	89
8.2	Overloading and qualified expressions	90
8.3	Character types	93
9	Access types (pointers)	95
9.1	Overview	95
9.2	Allocation (by type)	97
9.3	Dereferencing	98
9.4	Other features	98
9.5	Mutually recursive types	99
10	More about records	101
10.1	Dynamically sized record types	101
10.2	Records with discriminant	102
10.3	Variant records	104
11	Fixed-point types	107
11.1	Decimal fixed-point types	107
11.1.1	Decimal delta	107
11.1.2	Decimal digits	108
11.1.3	Decimal delta and digits	110
11.1.4	Requirements for the delta	111
11.2	Ordinary fixed-point types	111
11.3	Fixed-point vs. floating-point types	112
12	Privacy	115

12.1 Basic encapsulation	115
12.2 Abstract data types	116
12.3 Limited types	118
12.4 Child packages & privacy	119
13 Generics	125
13.1 Introduction	125
13.2 Formal type declaration	125
13.3 Formal object declaration	126
13.4 Generic body definition	126
13.5 Generic instantiation	127
13.6 Generic packages	128
13.7 Formal subprograms	130
13.8 Example: I/O instances	131
13.9 Example: ADTs	133
13.10 Example: Swap	135
13.11 Example: Reversing	138
13.12 Example: Test application	141
14 Exceptions	145
14.1 Exception declaration	145
14.2 Raising an exception	145
14.3 Handling an exception	146
14.4 Predefined exceptions	148
15 Tasking	149
15.1 Tasks	149
15.1.1 Simple task	149
15.1.2 Simple synchronization	150
15.1.3 Delay	152
15.1.4 Synchronization: rendezvous	153
15.1.5 Select loop	154
15.1.6 Cycling tasks	156
15.2 Protected objects	159
15.2.1 Simple object	160
15.2.2 Entries	161
15.3 Task and protected types	162
15.3.1 Task types	162
15.3.2 Protected types	164
16 Design by contracts	167
16.1 Pre- and postconditions	167
16.2 Predicates	170
16.3 Type invariants	174
17 Interfacing with C	177
17.1 Multi-language project	177
17.2 Type convention	177
17.3 Foreign subprograms	178
17.3.1 Calling C subprograms in Ada	178
17.3.2 Calling Ada subprograms in C	180
17.4 Foreign variables	181
17.4.1 Using C global variables in Ada	181
17.4.2 Using Ada variables in C	182
17.5 Generating bindings	184
17.5.1 Adapting bindings	185
18 Object-oriented programming	191
18.1 Derived types	192

18.2 Tagged types	193
18.3 Classwide types	195
18.4 Dispatching operations	196
18.5 Dot notation	198
18.6 Private & Limited	199
18.7 Classwide access types	201
19 Standard library: Containers	205
19.1 Vectors	205
19.1.1 Instantiation	205
19.1.2 Initialization	206
19.1.3 Appending and prepending elements	206
19.1.4 Accessing first and last elements	208
19.1.5 Iterating	209
19.1.6 Finding and changing elements	213
19.1.7 Inserting elements	215
19.1.8 Removing elements	216
19.1.9 Other Operations	219
19.2 Sets	221
19.2.1 Initialization and iteration	222
19.2.2 Operations on elements	223
19.2.3 Other Operations	225
19.3 Indefinite maps	227
19.3.1 Hashed maps	228
19.3.2 Ordered maps	229
19.3.3 Complexity	230
20 Standard library: Dates & Times	233
20.1 Date and time handling	233
20.1.1 Delaying using date	234
20.2 Real-time	237
20.2.1 Benchmarking	237
21 Standard library: Strings	241
21.1 String operations	241
21.2 Limitation of fixed-length strings	246
21.3 Bounded strings	246
21.4 Unbounded strings	249
22 Standard library: Files and streams	251
22.1 Text I/O	251
22.2 Sequential I/O	254
22.3 Direct I/O	256
22.4 Stream I/O	258
23 Standard library: Numerics	261
23.1 Elementary Functions	261
23.2 Random Number Generation	262
23.3 Complex Types	264
23.4 Vector and Matrix Manipulation	266
24 Appendices	271
24.1 Appendix A: Generic Formal Types	271
24.1.1 Indefinite version	272
24.2 Appendix B: Containers	273

II Advanced Journey With Ada: A Flight In Progress 275

25 Data types	279
25.1 Types	279
25.1.1 Names	279
25.1.2 Objects	283
25.1.3 Scalar Types	288
25.1.4 Enumerations	294
25.1.5 Universal and Root Types	302
25.1.6 Definite and Indefinite Subtypes	304
25.1.7 Incomplete types	310
25.1.8 Type view	312
25.1.9 Type conversion	319
25.1.10 Qualified Expressions	338
25.1.11 Default initial values	339
25.1.12 Deferred Constants	342
25.1.13 User-defined literals	344
25.2 Types and Representation	352
25.2.1 Enumeration Representation Clauses	352
25.2.2 Data Representation	353
25.2.3 Sizes	354
25.2.4 Alignment	363
25.2.5 Overlapping Storage	366
25.2.6 Packed Representation	368
25.2.7 Record Representation and storage clauses	373
25.2.8 Changing Data Representation	381
25.2.9 Valid Attribute	388
25.2.10 Unchecked Union	391
25.2.11 Addresses	397
25.2.12 Discarding names	405
25.3 Shared variable control	407
25.3.1 Volatile	408
25.3.2 Independent	410
25.3.3 Atomic	414
25.3.4 Full-access only	417
25.3.5 Atomic operations	430
25.4 Records	442
25.4.1 Default Initialization	442
25.4.2 Mutually dependent types	448
25.4.3 Null records	450
25.4.4 Record discriminants	458
25.4.5 Discriminant constraints and operations	482
25.4.6 Unknown discriminants	488
25.4.7 Unconstrained subtypes	496
25.4.8 Variant parts	498
25.4.9 Per-Object Expressions	511
25.5 Aggregates	516
25.5.1 Container Aggregates	516
25.5.2 Record aggregates	519
25.5.3 Full coverage rules for Aggregates	529
25.5.4 Array aggregates	531
25.5.5 Extension Aggregates	550
25.5.6 Delta Aggregates	557
25.6 Arrays	564
25.6.1 Array constraints	564
25.6.2 Multidimensional Arrays	567
25.6.3 Derived array types and array subtypes	575
25.7 Strings	579

25.7.1 Character and String Literals	579
25.7.2 Wide and Wide-Wide Strings	581
25.7.3 String Encoding	589
25.7.4 UTF-8 applications	595
25.7.5 Image attribute	605
25.7.6 Put_Image aspect	612
25.7.7 Universal text buffer	619
25.8 Numerics	622
25.8.1 Numeric Literals	622
25.8.2 Universal Numeric Types	629
25.8.3 Base types	643
25.8.4 Discrete and Real Numeric Types	648
25.8.5 Integer types	649
25.8.6 Modular Types	657
25.8.7 Floating-point types	660
25.8.8 Fixed-point types	679
25.8.9 Big Numbers	683
25.9 Numeric Attributes	698
25.9.1 Attributes of Modular Types	698
25.9.2 Attributes of Floating-Point Types	704
25.9.3 Attributes of Fixed-Point types	717
26 Control Flow	725
26.1 Expressions	725
26.1.1 Expressions: Definition	725
26.1.2 Conditional Expressions	731
26.1.3 Quantified Expressions	734
26.1.4 Declare Expressions	738
26.1.5 Reduction Expressions	742
26.2 Statements	748
26.2.1 Simple and Compound Statements	748
26.2.2 Labels	749
26.2.3 Exit loop statement	753
26.2.4 If, case and loop statements	754
26.2.5 Block Statements	758
26.2.6 Extended return statement	759
26.3 Subprograms	761
26.3.1 Parameter Modes and Associations	761
26.3.2 Operators	774
26.3.3 Expression functions	780
26.3.4 Overloading	783
26.3.5 Operator Overloading	788
26.3.6 Operator Overriding	788
26.3.7 Nonreturning procedures	791
26.3.8 Inline subprograms	794
26.3.9 Null Procedures	796
26.4 Exceptions	799
26.4.1 Classification of Errors	799
26.4.2 Asserts	803
26.4.3 Assertion policies	805
26.4.4 Checks and exceptions	808
26.4.5 Ada.Exceptions package	824
26.4.6 Exception renaming	832
26.4.7 Out and Uninitialized	833
26.4.8 Suppressing checks	837
27 Modular programming	841
27.1 Packages	841

27.1.1	Package renaming	841
27.1.2	Private packages	844
27.1.3	Private with clauses	852
27.1.4	Limited Visibility	855
27.1.5	Visibility	860
27.1.6	Use type clause	866
27.1.7	Use clauses and naming conflicts	869
27.2	Subprograms and Modularity	875
27.2.1	Private subprograms	875
28	Resource Management	881
28.1	Access Types	881
28.1.1	Access types: Terminology	881
28.1.2	Access types: Allocation	883
28.1.3	Discriminants as Access Values	891
28.1.4	Parameters as Access Values	898
28.1.5	Self-reference	908
28.1.6	Mutually dependent types using access types	911
28.1.7	Dereferencing	911
28.1.8	Ragged arrays	918
28.1.9	Aliasing	922
28.1.10	Accessibility Levels and Rules: An Introduction	932
28.1.11	Unchecked Access	942
28.1.12	Unchecked Deallocation	944
28.1.13	Null & Not Null Access	952
28.1.14	Design strategies for access types	956
28.1.15	Access to subprograms	964
28.1.16	Accessibility Rules and Access-To-Subprograms	989
28.1.17	Access and Address	993
28.2	Anonymous Access Types	997
28.2.1	Named and Anonymous Access Types	997
28.2.2	Anonymous Access-To-Object Types	1002
28.2.3	Access discriminants	1011
28.2.4	Self-reference	1018
28.2.5	Mutually dependent types using anonymous access types	1021
28.2.6	Access parameters	1021
28.2.7	User-Defined References	1030
28.2.8	Anonymous Access Types and Accessibility Rules	1039
28.2.9	Anonymous Access-To-Subprograms	1044
28.2.10	Accessibility Rules and Anonymous Access-To-Subprograms	1053
28.3	Limited Types	1063
28.3.1	Assignment and equality	1063
28.3.2	Limited private types	1071
28.3.3	Explicitly limited types	1079
28.3.4	Subtypes of Limited Types	1080
28.3.5	Deriving from limited types	1081
28.3.6	Immutably Limited Types	1089
28.3.7	Limited Types with Discriminants	1091
28.3.8	Record components of limited type	1097
28.3.9	Limited types and aggregates	1098
28.3.10	Constructor functions for limited types	1102
28.3.11	Return objects	1106
28.3.12	Building objects from constructors	1112
28.3.13	Limited types as parameter	1116
28.4	Controlled Types	1118
28.4.1	Overview	1118
28.4.2	Initialization	1130
28.4.3	Assignment	1141

28.4.4 Finalization	1147
28.4.5 Controlled Types and Exception Handling	1156
28.4.6 Applications of Controlled Types	1164
III Ada In Practice	1175
29 Introduction	1179
29.1 Assumptions	1180
29.2 Definitions	1180
29.2.1 Suppliers and Clients	1180
29.2.2 Compile-time Visibility	1180
29.2.3 Views	1181
29.2.4 Partial and Full Views	1181
30 Essential Design Idioms for Packages	1183
30.1 Motivation	1183
30.2 Implementation(s)	1183
30.2.1 Named Collection of Declarations	1184
30.2.2 Groups of Related Program Units	1184
30.3 Notes	1186
31 Abstract Data Types	1187
31.1 Motivation	1187
31.2 Implementation(s)	1188
31.3 Pros	1191
31.4 Cons	1191
31.5 Relationship With Other Idioms	1192
31.6 Notes	1192
32 Abstract Data Machines	1193
32.1 Motivation	1193
32.2 Implementation(s)	1193
32.3 Pros	1197
32.4 Cons	1197
33 Programming by Extension	1199
33.1 Motivation	1199
33.2 Implementation(s)	1200
33.3 Pros	1201
33.4 Cons	1201
33.5 Relationship With Other Idioms	1202
33.6 Notes	1202
34 Constructor Functions For Abstract Data Types	1203
34.1 Motivation	1203
34.2 Implementation(s)	1205
34.3 Pros	1206
34.4 Cons	1207
34.5 Relationship With Other Idioms	1207
34.6 Notes	1207
35 Controlling Object Initialization and Creation	1211
35.1 Motivation	1211
35.2 Implementation(s)	1213
35.2.1 Compile-Time Legality	1213
35.2.2 Run-Time Checks	1218
35.2.3 Preventing Object Creation by Clients	1222
35.3 Pros	1228

35.4 Cons	1228
35.5 Relationship With Other Idioms	1228
35.6 Notes	1228
36Type Punning	1229
36.1 Motivation	1229
36.2 Implementation(s)	1230
36.2.1 Overlays	1230
36.2.2 Unchecked Conversions on Address Values	1231
36.3 Pros	1235
36.4 Cons	1235
36.5 Relationship With Other Idioms	1235
36.6 Notes	1235
37Using Building Blocks to Express Inheritance Idioms	1237
37.1 Motivation	1237
37.1.1 Building Blocks	1239
37.2 Implementation(s)	1240
37.2.1 Subtype Inheritance	1241
37.2.2 Implementation Inheritance	1244
37.3 Pros	1245
37.4 Cons	1246
37.5 Relationship With Other Idioms	1246
38Providing Component Access to Enclosing Record Objects	1247
38.1 Motivation	1247
38.2 Implementation(s)	1248
38.3 Real-World Example	1250
38.4 Pros	1254
38.5 Cons	1254
38.6 Relationship With Other Idioms	1254
38.7 Notes	1254
38.8 Full Source Code for Selected Units	1255
39Interrupt Handling	1261
39.1 Motivation	1261
39.2 Implementation(s)	1262
39.2.1 First Level Handler Alone	1262
39.2.2 Task Notification Introduction	1265
39.2.3 Task Notification With Communication	1266
39.2.4 Task Notification Without Communication	1269
39.3 Pros	1273
39.4 Cons	1273
39.5 Relationship With Other Idioms	1273
39.6 What About Priorities?	1273
39.7 Notes	1274
40Reducing Object Code from Generic Package Instantiations	1275
40.1 Motivation	1275
40.2 Implementation(s)	1276
40.3 Pros	1276
40.4 Cons	1277
40.5 Relationship With Other Idioms	1277
40.6 Notes	1277
41Resource Acquisition Is Initialization (RAII)	1279
41.1 Motivation	1279
41.2 Implementation	1282
41.3 Pros	1285

41.4 Cons	1285
41.5 Relationship With Other Idioms	1285
41.6 Notes	1285
42 Using Streams for API Flexibility	1287
42.1 Motivation	1287
42.2 Implementation	1287
42.3 Pros	1295
42.4 Cons	1295
42.5 Relationship With Other Idioms	1296
42.6 Notes	1296
43 Dealing with Silent Task Termination	1297
43.1 Motivation	1297
43.2 Implementation	1297
43.3 Pros	1308
43.4 Cons	1308
43.5 Relationship With Other Idioms	1308
43.6 Notes	1309
44 Idioms for Protected Objects	1311
44.1 Motivation	1313
44.2 Implementation	1314
44.2.1 First Idiom Category: Application-Specific Protected Objects	1315
44.2.2 Second Idiom Category: Developer-Defined Concurrency Abstractions	1315
44.2.3 Concurrent Programming	1331
44.2.4 Real-Time Programming	1331
44.2.5 Embedded Systems Programming	1332
44.3 Pros	1332
44.4 Cons	1333
44.5 Relationship With Other Idioms	1333
44.6 Notes	1333
44.7 Full Code for Event_Management Generic Package	1333
 IV Ada for the C++ or Java Developer	 1339
45 Preface	1343
46 Basics	1345
47 Compilation Unit Structure	1347
48 Statements, Declarations, and Control Structures	1349
48.1 Statements and Declarations	1349
48.2 Conditions	1351
48.3 Loops	1352
49 Type System	1355
49.1 Strong Typing	1355
49.2 Language-Defined Types	1356
49.3 Application-Defined Types	1356
49.4 Type Ranges	1358
49.5 Generalized Type Contracts: Subtype Predicates	1359
49.6 Attributes	1359
49.7 Arrays and Strings	1360
49.8 Heterogeneous Data Structures	1363
49.9 Pointers	1364

50 Functions and Procedures	1367
50.1 General Form	1367
50.2 Overloading	1368
50.3 Subprogram Contracts	1369
51 Packages	1371
51.1 Declaration Protection	1371
51.2 Hierarchical Packages	1372
51.3 Using Entities from Packages	1372
52 Classes and Object Oriented Programming	1375
52.1 Primitive Subprograms	1375
52.2 Derivation and Dynamic Dispatch	1376
52.3 Constructors and Destructors	1379
52.4 Encapsulation	1379
52.5 Abstract Types and Interfaces	1380
52.6 Invariants	1382
53 Generics	1385
53.1 Generic Subprograms	1385
53.2 Generic Packages	1386
53.3 Generic Parameters	1387
54 Exceptions	1389
54.1 Standard Exceptions	1389
54.2 Custom Exceptions	1390
55 Concurrency	1391
55.1 Tasks	1391
55.2 Rendezvous	1394
55.3 Selective Rendezvous	1396
55.4 Protected Objects	1397
56 Low Level Programming	1401
56.1 Representation Clauses	1401
56.2 Embedded Assembly Code	1402
56.3 Interfacing with C	1403
57 Conclusion	1405
58 References	1407
 V Introduction To SPARK	 1409
59 SPARK Overview	1413
59.1 What is it?	1413
59.2 What do the tools do?	1413
59.3 Key Tools	1414
59.4 A trivial example	1414
59.5 The Programming Language	1415
59.6 Limitations	1415
59.6.1 No side-effects in expressions	1415
59.6.2 No aliasing of names	1418
59.7 Designating SPARK Code	1420
59.8 Code Examples / Pitfalls	1421
59.8.1 Example #1	1421
59.8.2 Example #2	1422
59.8.3 Example #3	1423
59.8.4 Example #4	1424

59.8.5 Example #5	1425
59.8.6 Example #6	1426
59.8.7 Example #7	1427
59.8.8 Example #8	1427
59.8.9 Example #9	1428
59.8.10 Example #10	1429
60 Flow Analysis	1431
60.1 What does flow analysis do?	1431
60.2 Errors Detected	1431
60.2.1 Uninitialized Variables	1431
60.2.2 Ineffective Statements	1432
60.2.3 Incorrect Parameter Mode	1433
60.3 Additional Verifications	1434
60.3.1 Global Contracts	1434
60.3.2 Depends Contracts	1436
60.4 Shortcomings	1438
60.4.1 Modularity	1438
60.4.2 Composite Types	1438
60.4.3 Value Dependency	1441
60.4.4 Contract Computation	1442
60.5 Code Examples / Pitfalls	1443
60.5.1 Example #1	1443
60.5.2 Example #2	1444
60.5.3 Example #3	1445
60.5.4 Example #4	1446
60.5.5 Example #5	1448
60.5.6 Example #6	1449
60.5.7 Example #7	1450
60.5.8 Example #8	1451
60.5.9 Example #9	1452
60.5.10 Example #10	1454
61 Proof of Program Integrity	1457
61.1 Runtime Errors	1457
61.2 Modularity	1459
61.2.1 Exceptions	1460
61.3 Contracts	1462
61.3.1 Executable Semantics	1464
61.3.2 Additional Assertions and Contracts	1465
61.4 Debugging Failed Proof Attempts	1467
61.4.1 Debugging Errors in Code or Specification	1467
61.4.2 Debugging Cases where more Information is Required	1469
61.4.3 Debugging Prover Limitations	1470
61.5 Code Examples / Pitfalls	1473
61.5.1 Example #1	1473
61.5.2 Example #2	1474
61.5.3 Example #3	1475
61.5.4 Example #4	1476
61.5.5 Example #5	1477
61.5.6 Example #6	1478
61.5.7 Example #7	1479
61.5.8 Example #8	1481
61.5.9 Example #9	1482
61.5.10 Example #10	1483
62 State Abstraction	1485
62.1 What's an Abstraction?	1485
62.2 Why is Abstraction Useful?	1486

62.3	Abstraction of a Package's State	1487
62.4	Declaring a State Abstraction	1487
62.5	Refining an Abstract State	1488
62.6	Representing Private Variables	1489
62.7	Additional State	1490
62.7.1	Nested Packages	1490
62.7.2	Constants that Depend on Variables	1491
62.8	Subprogram Contracts	1493
62.8.1	Global and Depends	1493
62.8.2	Preconditions and Postconditions	1495
62.9	Initialization of Local Variables	1498
62.10	Code Examples / Pitfalls	1500
62.10.1	Example #1	1500
62.10.2	Example #2	1501
62.10.3	Example #3	1502
62.10.4	Example #4	1503
62.10.5	Example #5	1504
62.10.6	Example #6	1505
62.10.7	Example #7	1506
62.10.8	Example #8	1508
62.10.9	Example #9	1510
62.10.10	Example #10	1511
63	Proof of Functional Correctness	1513
63.1	Beyond Program Integrity	1513
63.2	Advanced Contracts	1516
63.2.1	Ghost Code	1517
63.2.2	Ghost Functions	1520
63.2.3	Global Ghost Variables	1522
63.3	Guide Proof	1524
63.3.1	Local Ghost Variables	1524
63.3.2	Ghost Procedures	1526
63.3.3	Handling of Loops	1527
63.3.4	Loop Invariants	1529
63.4	Code Examples / Pitfalls	1534
63.4.1	Example #1	1534
63.4.2	Example #2	1536
63.4.3	Example #3	1537
63.4.4	Example #4	1539
63.4.5	Example #5	1540
63.4.6	Example #6	1541
63.4.7	Example #7	1543
63.4.8	Example #8	1544
63.4.9	Example #9	1545
63.4.10	Example #10	1546
VI	SPARK Ada for the MISRA C Developer	1549
64	Preface	1553
65	Enforcing Basic Program Consistency	1555
65.1	Taming Text-Based Inclusion	1555
65.2	Hardening Link-Time Checking	1558
65.3	Going Towards Encapsulation	1560
66	Enforcing Basic Syntactic Guarantees	1563
66.1	Distinguishing Code and Comments	1563
66.2	Specially Handling Function Parameters and Result	1564

66.2.1 Handling the Result of Function Calls	1564
66.2.2 Handling Function Parameters	1565
66.3 Ensuring Control Structures Are Not Abused	1566
66.3.1 Preventing the Semicolon Mistake	1566
66.3.2 Avoiding Complex Switch Statements	1567
66.3.3 Avoiding Complex Loops	1570
66.3.4 Avoiding the Dangling Else Issue	1571
67 Enforcing Strong Typing	1575
67.1 Enforcing Strong Typing for Pointers	1575
67.1.1 Pointers Are Not Addresses	1576
67.1.2 Pointers Are Not References	1576
67.1.3 Pointers Are Not Arrays	1577
67.1.4 Pointers Should Be Typed	1581
67.2 Enforcing Strong Typing for Scalars	1583
67.2.1 Restricting Operations on Types	1583
67.2.2 Restricting Explicit Conversions	1588
67.2.3 Restricting Implicit Conversions	1588
68 Initializing Data Before Use	1593
68.1 Detecting Reads of Uninitialized Data	1593
68.2 Detecting Partial or Redundant Initialization of Arrays and Structures	1598
69 Controlling Side Effects	1603
69.1 Preventing Undefined Behavior	1603
69.2 Reducing Programmer Confusion	1603
69.3 Side Effects and SPARK	1604
70 Detecting Undefined Behavior	1609
70.1 Preventing Undefined Behavior in SPARK	1609
70.2 Proof of Absence of Run-Time Errors in SPARK	1610
71 Detecting Unreachable Code and Dead Code	1615
72 Conclusion	1619
73 References	1621
73.1 About MISRA C	1621
73.2 About SPARK	1621
73.3 About MISRA C and SPARK	1622
VII Introduction to Embedded Systems Programming	1623
74 Introduction	1627
74.1 So, what will we actually cover?	1627
74.2 Definitions	1628
74.3 Down To The Bare Metal	1628
74.4 The Ada Drivers Library	1628
75 Low Level Programming	1631
75.1 Separation Principle	1631
75.2 Guaranteed Level of Support	1632
75.3 Querying Implementation Limits and Characteristics	1633
75.4 Querying Representation Choices	1636
75.5 Specifying Representation	1641
75.6 Unchecked Programming	1655
75.7 Data Validity	1664

76 Multi-Language Development	1667
76.1 General Interfacing	1668
76.1.1 Aspect/Pragma Convention	1668
76.1.2 Aspect/Pragma Import and Export	1671
76.1.3 Aspect/Pragma External_Name and Link_Name	1672
76.1.4 Package Interfaces	1673
76.2 Language-Specific Interfacing	1675
76.2.1 Package Interfaces.C	1675
76.2.2 Package Interfaces.C.Strings	1680
76.2.3 Package Interfaces.C.Pointers	1681
76.2.4 Package Interfaces.Fortran	1681
76.2.5 Machine Code Insertions (MCI)	1681
76.3 When Ada Is Not the Main Language	1686
77 Interacting with Devices	1689
77.1 Non-Memory-Mapped Devices	1690
77.2 Memory-Mapped Devices	1691
77.3 Dynamic Address Conversion	1699
77.4 Address Arithmetic	1701
78 General-Purpose Code Generators	1703
78.1 Aspect Independent	1703
78.2 Aspect Volatile	1706
78.3 Aspect Atomic	1709
78.4 Aspect Full_Access_Only	1710
79 Handling Interrupts	1715
79.1 Background	1715
79.2 Language-Defined Interrupt Model	1719
79.3 Interrupt Handlers	1720
79.4 Interrupt Management	1723
79.5 Associating Handlers With Interrupts	1723
79.6 Interrupt Priorities	1726
79.7 Common Design Idioms	1728
79.7.1 Parameterizing Handlers	1728
79.7.2 Multi-Level Handlers	1730
79.8 Final Points	1736
80 Conclusion	1737
VIII Ada for the Embedded C Developer	1739
81 Introduction	1743
81.1 So, what is this Ada thing anyway?	1743
81.2 Ada — The Technical Details	1745
82 The C Developer's Perspective on Ada	1747
82.1 What we mean by Embedded Software	1747
82.2 The GNAT Toolchain	1747
82.3 The GNAT Toolchain for Embedded Targets	1747
82.4 Hello World in Ada	1748
82.5 The Ada Syntax	1749
82.6 Compilation Unit Structure	1750
82.7 Packages	1751
82.7.1 Declaration Protection	1751
82.7.2 Hierarchical Packages	1752
82.7.3 Using Entities from Packages	1752
82.8 Statements and Declarations	1753

82.9 Conditions	1759
82.10 Loops	1763
82.11 Type System	1769
82.11.1 Strong Typing	1769
82.11.2 Language-Defined Types	1773
82.11.3 Application-Defined Types	1773
82.11.4 Type Ranges	1776
82.11.5 Unsigned And Modular Types	1779
82.11.6 Attributes	1782
82.11.7 Arrays and Strings	1784
82.11.8 Heterogeneous Data Structures	1792
82.11.9 Pointers	1793
82.12 Functions and Procedures	1799
82.12.1 General Form	1799
82.12.2 Overloading	1802
82.12.3 Aspects	1804
83 Concurrency and Real-Time	1807
83.1 Understanding the various options	1807
83.2 Tasks	1807
83.3 Rendezvous	1810
83.4 Selective Rendezvous	1811
83.5 Protected Objects	1814
83.6 Ravenscar	1817
84 Writing Ada on Embedded Systems	1821
84.1 Understanding the Ada Run-Time	1821
84.2 Low Level Programming	1822
84.2.1 Representation Clauses	1822
84.2.2 Embedded Assembly Code	1823
84.3 Interrupt Handling	1824
84.4 Dealing with Absence of FPU with Fixed Point	1826
84.5 Volatile and Atomic data	1830
84.5.1 Volatile	1831
84.5.2 Atomic	1833
84.6 Interfacing with Devices	1834
84.6.1 Size aspect and attribute	1835
84.6.2 Register overlays	1836
84.6.3 Data streams	1839
84.7 ARM and svd2ada	1844
85 Enhancing Verification with SPARK and Ada	1845
85.1 Understanding Exceptions and Dynamic Checks	1845
85.2 Understanding Dynamic Checks versus Formal Proof	1852
85.3 Initialization and Correct Data Flow	1855
85.4 Contract-Based Programming	1856
85.5 Replacing Defensive Code	1859
85.6 Proving Absence of Run-Time Errors	1861
85.7 Proving Abstract Properties	1862
85.8 Final Comments	1863
86 C to Ada Translation Patterns	1865
86.1 Naming conventions and casing considerations	1865
86.2 Manually interfacing C and Ada	1865
86.3 Building and Debugging mixed language code	1867
86.4 Automatic interfacing	1868
86.5 Using Arrays in C interfaces	1868
86.6 By-value vs. by-reference types	1870
86.7 Naming and prefixes	1871

86.8 Pointers	1872
86.9 Bitwise Operations	1875
86.10 Mapping Structures to Bit-Fields	1878
86.10.1 Overlays vs. Unchecked Conversions	1891
87 Handling Variability and Re-usability	1895
87.1 Understanding static and dynamic variability	1895
87.2 Handling variability & reusability statically	1895
87.2.1 Genericity	1895
87.2.2 Simple derivation	1898
87.2.3 Configuration pragma files	1904
87.2.4 Configuration packages	1906
87.3 Handling variability & reusability dynamically	1910
87.3.1 Records with discriminants	1910
87.3.2 Variant records	1912
87.3.3 Object orientation	1920
87.3.4 Pointer to subprograms	1933
87.4 Design by components using dynamic libraries	1938
88 Performance considerations	1943
88.1 Overall expectations	1943
88.2 Switches and optimizations	1943
88.2.1 Optimizations levels	1943
88.2.2 Inlining	1944
88.3 Checks and assertions	1945
88.3.1 Checks	1945
88.3.2 Assertions	1949
88.4 Dynamic vs. static structures	1949
88.5 Pointers vs. data copies	1950
88.5.1 Function returns	1953
89 Argumentation and Business Perspectives	1957
89.1 What's the expected ROI of a C to Ada transition?	1957
89.2 Who is using Ada today?	1958
89.3 What is the future of the Ada technology?	1958
89.4 Is the Ada toolset complete?	1959
89.5 Where can I find Ada or SPARK developers?	1959
89.6 How to introduce Ada and SPARK in an existing code base?	1959
90 Conclusion	1961
91 Appendix A: Hands-On Object-Oriented Programming	1965
91.1 System Overview	1965
91.2 Non Object-Oriented Approach	1966
91.2.1 Starting point in C	1966
91.2.2 Initial translation to Ada	1969
91.2.3 Improved Ada implementation	1973
91.3 First Object-Oriented Approach	1977
91.3.1 Interfaces	1977
91.3.2 Base type	1977
91.3.3 Derived types	1978
91.3.4 Subprograms from parent	1979
91.3.5 Type AB	1980
91.3.6 Updated source-code	1980
91.4 Further Improvements	1984
91.4.1 Dispatching calls	1984
91.4.2 Dynamic allocation	1985
91.4.3 Limited controlled types	1986
91.4.4 Updated source-code	1987

IX Guidelines for Safe and Secure Ada/SPARK

1993

92 Introduction

1997

92.1 Scope	1997
92.2 Structure	1997
92.3 Enforcement	1997
92.4 About the Rules	1998
92.4.1 Mapping to Other Standards	1998

93 Definitions

1999

93.1 Level	1999
93.2 Remediation	1999

94 Dynamic Storage Management (DYN)

2001

94.1 Common High Integrity Restrictions (DYN01)	2002
94.1.1 Reference	2002
94.1.2 Description	2002
94.1.3 Applicable Vulnerability within ISO TR 24772-2	2003
94.1.4 Applicable Common Weakness Enumeration	2003
94.1.5 Noncompliant Code Example	2003
94.1.6 Compliant Code Example	2003
94.1.7 Notes	2004
94.2 Traditional Static Allocation Policy (DYN02)	2004
94.2.1 Reference	2004
94.2.2 Description	2004
94.2.3 Applicable Vulnerability within ISO TR 24772-2	2005
94.2.4 Applicable Common Weakness Enumeration	2005
94.2.5 Noncompliant Code Example	2005
94.2.6 Compliant Code Example	2005
94.2.7 Notes	2005
94.3 Access Types Without Allocators Policy (DYN03)	2005
94.3.1 Reference	2006
94.3.2 Description	2006
94.3.3 Applicable Vulnerability within ISO TR 24772-2	2006
94.3.4 Applicable Common Weakness Enumeration	2006
94.3.5 Noncompliant Code Example	2007
94.3.6 Compliant Code Example	2007
94.3.7 Notes	2007
94.4 Minimal Dynamic Allocation Policy (DYN04)	2007
94.4.1 Reference	2007
94.4.2 Description	2007
94.4.3 Applicable Vulnerability within ISO TR 24772-2	2008
94.4.4 Applicable Common Weakness Enumeration	2008
94.4.5 Noncompliant Code Example	2008
94.4.6 Compliant Code Example	2008
94.4.7 Notes	2008
94.5 User-Defined Storage Pools Policy (DYN05)	2008
94.5.1 Reference	2009
94.5.2 Description	2009
94.5.3 Applicable Vulnerability within ISO TR 24772-2	2009
94.5.4 Applicable Common Weakness Enumeration	2009
94.5.5 Noncompliant Code Example	2010
94.5.6 Compliant Code Example	2010
94.5.7 Notes	2010
94.6 Statically Determine Maximum Stack Requirements (DYN06)	2010
94.6.1 Reference	2011
94.6.2 Description	2011
94.6.3 Applicable Vulnerability within ISO TR 24772-2	2011
94.6.4 Applicable Common Weakness Enumeration	2011

94.6.5 Noncompliant Code Example	2011
94.6.6 Compliant Code Example	2011
94.6.7 Notes	2011
95 Safe Reclamation (RCL)	2013
95.1 No Multiple Reclamations (RCL01)	2013
95.1.1 Reference	2014
95.1.2 Description	2014
95.1.3 Applicable Vulnerability within ISO TR 24772-2	2014
95.1.4 Applicable Common Weakness Enumeration	2014
95.1.5 Noncompliant Code Example	2014
95.1.6 Compliant Code Example	2014
95.1.7 Notes	2014
95.2 Only Reclaim Allocated Storage (RCL02)	2014
95.2.1 Reference	2015
95.2.2 Description	2015
95.2.3 Applicable Vulnerability within ISO TR 24772-2	2015
95.2.4 Applicable Common Weakness Enumeration	2015
95.2.5 Noncompliant Code Example	2015
95.2.6 Compliant Code Example	2015
95.2.7 Notes	2015
95.3 Only Reclaim to the Same Pool (RCL03)	2016
95.3.1 Reference	2016
95.3.2 Description	2016
95.3.3 Applicable Vulnerability within ISO TR 24772-2	2016
95.3.4 Applicable Common Weakness Enumeration	2016
95.3.5 Noncompliant Code Example	2016
95.3.6 Compliant Code Example	2017
95.3.7 Notes	2017
96 Concurrency (CON)	2019
96.1 Use the Ravenscar Profile (CON01)	2020
96.1.1 Reference	2020
96.1.2 Description	2021
96.1.3 Applicable Vulnerability within ISO TR 24772-2	2021
96.1.4 Applicable Common Weakness Enumeration	2021
96.1.5 Noncompliant Code Example	2022
96.1.6 Compliant Code Example	2022
96.1.7 Notes	2022
96.2 Use the Jorvik Profile (CON02)	2022
96.2.1 Reference	2023
96.2.2 Description	2023
96.2.3 Applicable Vulnerability within ISO TR 24772-2	2024
96.2.4 Applicable Common Weakness Enumeration	2024
96.2.5 Noncompliant Code Example	2024
96.2.6 Compliant Code Example	2024
96.2.7 Notes	2024
96.3 Avoid Shared Variables for Inter-task Communication (CON03)	2025
96.3.1 Reference	2025
96.3.2 Description	2025
96.3.3 Applicable Vulnerability within ISO TR 24772-2	2025
96.3.4 Applicable Common Weakness Enumeration	2025
96.3.5 Noncompliant Code Example	2026
96.3.6 Compliant Code Example	2026
96.3.7 Notes	2026
97 Robust Programming Practice (RPP)	2027
97.1 No Use of "others" in Case Constructs (RPP01)	2027
97.1.1 Reference	2028

97.1.2 Description	2028
97.1.3 Applicable Vulnerability within ISO TR 24772-2	2028
97.1.4 Applicable Common Weakness Enumeration	2028
97.1.5 Noncompliant Code Example	2028
97.1.6 Compliant Code Example	2028
97.1.7 Notes	2028
97.2 No Enumeration Ranges in Case Constructs (RPP02)	2029
97.2.1 Reference	2029
97.2.2 Description	2029
97.2.3 Applicable Vulnerability within ISO TR 24772-2	2029
97.2.4 Applicable Common Weakness Enumeration	2029
97.2.5 Noncompliant Code Example	2029
97.2.6 Compliant Code Example	2030
97.2.7 Notes	2030
97.3 Limited Use of "others" in Aggregates (RPP03)	2030
97.3.1 Reference	2030
97.3.2 Description	2030
97.3.3 Applicable Vulnerability within ISO TR 24772-2	2031
97.3.4 Applicable Common Weakness Enumeration	2031
97.3.5 Noncompliant Code Example	2031
97.3.6 Compliant Code Example	2031
97.3.7 Notes	2031
97.4 No Unassigned Mode-Out Procedure Parameters (RPP04)	2031
97.4.1 Reference	2032
97.4.2 Description	2032
97.4.3 Applicable Vulnerability within ISO TR 24772-2	2032
97.4.4 Applicable Common Weakness Enumeration	2032
97.4.5 Noncompliant Code Example	2032
97.4.6 Compliant Code Example	2033
97.4.7 Notes	2033
97.5 No Use of "others" in Exception Handlers (RPP05)	2033
97.5.1 Reference	2033
97.5.2 Description	2034
97.5.3 Applicable Vulnerability within ISO TR 24772-2	2034
97.5.4 Applicable Common Weakness Enumeration	2034
97.5.5 Noncompliant Code Example	2034
97.5.6 Compliant Code Example	2034
97.5.7 Notes	2034
97.6 Avoid Function Side-Effects (RPP06)	2034
97.6.1 Reference	2035
97.6.2 Description	2035
97.6.3 Applicable Vulnerability within ISO TR 24772-2	2035
97.6.4 Applicable Common Weakness Enumeration	2035
97.6.5 Noncompliant Code Example	2035
97.6.6 Compliant Code Example	2036
97.6.7 Notes	2036
97.7 Functions Only Have Mode "in" (RPP07)	2036
97.7.1 Reference	2036
97.7.2 Description	2036
97.7.3 Applicable Vulnerability within ISO TR 24772-2	2037
97.7.4 Applicable Common Weakness Enumeration	2037
97.7.5 Noncompliant Code Example	2037
97.7.6 Compliant Code Example	2037
97.7.7 Notes	2037
97.8 Limit Parameter Aliasing (RPP08)	2037
97.8.1 Reference	2038
97.8.2 Description	2038
97.8.3 Applicable Vulnerability within ISO TR 24772-2	2039

97.8.4 Applicable Common Weakness Enumeration	2039
97.8.5 Noncompliant Code Example	2039
97.8.6 Compliant Code Example	2039
97.8.7 Notes	2039
97.9 Use Precondition and Postcondition Contracts (RPP09)	2039
97.9.1 Reference	2040
97.9.2 Description	2040
97.9.3 Applicable Vulnerability within ISO TR 24772-2	2040
97.9.4 Applicable Common Weakness Enumeration	2040
97.9.5 Noncompliant Code Example	2040
97.9.6 Compliant Code Example	2040
97.9.7 Notes	2041
97.10 Do Not Re-Verify Preconditions in Subprogram Bodies (RPP10)	2041
97.10.1 Reference	2041
97.10.2 Description	2041
97.10.3 Applicable Vulnerability within ISO TR 24772-2	2041
97.10.4 Applicable Common Weakness Enumeration	2042
97.10.5 Noncompliant Code Example	2042
97.10.6 Compliant Code Example	2042
97.10.7 Notes	2042
97.11 Always Use the Result of Function Calls (RPP11)	2042
97.11.1 Reference	2043
97.11.2 Description	2043
97.11.3 Applicable Vulnerability within ISO TR 24772-2	2043
97.11.4 Applicable Common Weakness Enumeration	2043
97.11.5 Noncompliant Code Example	2043
97.11.6 Compliant Code Example	2043
97.11.7 Notes	2043
97.12 No Recursion (RPP12)	2044
97.12.1 Reference	2044
97.12.2 Description	2044
97.12.3 Applicable Vulnerability within ISO TR 24772-2	2044
97.12.4 Applicable Common Weakness Enumeration	2044
97.12.5 Noncompliant Code Example	2044
97.12.6 Compliant Code Example	2045
97.12.7 Notes	2045
97.13 No Reuse of Standard Typemarks (RPP13)	2045
97.13.1 Reference	2045
97.13.2 Description	2045
97.13.3 Applicable Vulnerability within ISO TR 24772-2	2046
97.13.4 Applicable Common Weakness Enumeration	2046
97.13.5 Noncompliant Code Example	2046
97.13.6 Compliant Code Example	2046
97.13.7 Notes	2046
97.14 Use Symbolic Constants for Literal Values (RPP14)	2046
97.14.1 Reference	2047
97.14.2 Description	2047
97.14.3 Applicable Vulnerability within ISO TR 24772-2	2047
97.14.4 Applicable Common Weakness Enumeration	2047
97.14.5 Noncompliant Code Example	2047
97.14.6 Compliant Code Example	2047
97.14.7 Notes	2047
98 Exception Usage (EXU)	2049
98.1 Do Not Raise Language-Defined Exceptions (EXU01)	2050
98.1.1 Reference	2050
98.1.2 Description	2050
98.1.3 Applicable Vulnerability within ISO TR 24772-2	2050

98.1.4 Applicable Common Weakness Enumeration	2050
98.1.5 Noncompliant Code Example	2050
98.1.6 Compliant Code Example	2051
98.1.7 Notes	2051
98.2 No Unhandled Application-Defined Exceptions (EXU02)	2051
98.2.1 Reference	2052
98.2.2 Description	2052
98.2.3 Applicable Vulnerability within ISO TR 24772-2	2052
98.2.4 Applicable Common Weakness Enumeration	2052
98.2.5 Noncompliant Code Example	2053
98.2.6 Compliant Code Example	2053
98.2.7 Notes	2053
98.3 No Exception Propagation Beyond Name Visibility (EXU03)	2053
98.3.1 Reference	2054
98.3.2 Description	2054
98.3.3 Applicable Vulnerability within ISO TR 24772-2	2054
98.3.4 Applicable Common Weakness Enumeration	2054
98.3.5 Noncompliant Code Example	2054
98.3.6 Compliant Code Example	2054
98.3.7 Notes	2055
98.4 Prove Absence of Run-time Exceptions (EXU04)	2055
98.4.1 Reference	2055
98.4.2 Description	2055
98.4.3 Applicable Vulnerability within ISO TR 24772-2	2056
98.4.4 Applicable Common Weakness Enumeration	2056
98.4.5 Noncompliant Code Example	2056
98.4.6 Compliant Code Example	2056
98.4.7 Notes	2056
99 Object-Oriented Programming (OOP)	2057
99.1 No Class-wide Constructs Policy (OOP01)	2057
99.1.1 Reference	2058
99.1.2 Description	2058
99.1.3 Applicable Vulnerability within ISO TR 24772-2	2058
99.1.4 Applicable Common Weakness Enumeration	2058
99.1.5 Noncompliant Code Example	2058
99.1.6 Compliant Code Example	2058
99.1.7 Notes	2058
99.2 Static Dispatching Only Policy (OOP02)	2059
99.2.1 Reference	2059
99.2.2 Description	2059
99.2.3 Applicable Vulnerability within ISO TR 24772-2	2059
99.2.4 Applicable Common Weakness Enumeration	2059
99.2.5 Noncompliant Code Example	2059
99.2.6 Compliant Code Example	2060
99.2.7 Notes	2060
99.3 Limit Inheritance Hierarchy Depth (OOP03)	2060
99.3.1 Reference	2060
99.3.2 Description	2060
99.3.3 Applicable Vulnerability within ISO TR 24772-2	2061
99.3.4 Applicable Common Weakness Enumeration	2061
99.3.5 Noncompliant Code Example	2061
99.3.6 Compliant Code Example	2061
99.3.7 Notes	2061
99.4 Limit Statically-Dispatched Calls to Primitive Operations (OOP04)	2061
99.4.1 Reference	2062
99.4.2 Description	2062
99.4.3 Applicable Vulnerability within ISO TR 24772-2	2062

99.4.4 Applicable Common Weakness Enumeration	2063
99.4.5 Noncompliant Code Example	2063
99.4.6 Compliant Code Example	2063
99.4.7 Notes	2063
99.5 Use Explicit Overriding Annotations (OOP05)	2063
99.5.1 Reference	2064
99.5.2 Description	2064
99.5.3 Applicable Vulnerability within ISO TR 24772-2	2065
99.5.4 Applicable Common Weakness Enumeration	2065
99.5.5 Noncompliant Code Example	2065
99.5.6 Compliant Code Example	2065
99.5.7 Notes	2065
99.6 Use Class-wide Pre/Post Contracts (OOP06)	2065
99.6.1 Reference	2066
99.6.2 Description	2066
99.6.3 Applicable Vulnerability within ISO TR 24772-2	2066
99.6.4 Applicable Common Weakness Enumeration	2066
99.6.5 Noncompliant Code Example	2066
99.6.6 Compliant Code Example	2066
99.6.7 Notes	2067
99.7 Ensure Local Type Consistency (OOP07)	2067
99.7.1 Reference	2067
99.7.2 Description	2067
99.7.3 Applicable Vulnerability within ISO TR 24772-2	2069
99.7.4 Applicable Common Weakness Enumeration	2069
99.7.5 Noncompliant Code Example	2069
99.7.6 Compliant Code Example	2070
99.7.7 Notes	2071
100 Software Engineering (SWE)	2073
100.1 Use SPARK Extensively (SWE01)	2073
100.1.1 Reference	2074
100.1.2 Description	2074
100.1.3 Applicable Vulnerability within ISO TR 24772-2	2074
100.1.4 Applicable Common Weakness Enumeration	2074
100.1.5 Noncompliant Code Example	2074
100.1.6 Compliant Code Example	2074
100.1.7 Notes	2074
100.2 Enable Optional Warnings and Treat As Errors (SWE02)	2074
100.2.1 Reference	2075
100.2.2 Description	2075
100.2.3 Applicable Vulnerability within ISO TR 24772-2	2075
100.2.4 Applicable Common Weakness Enumeration	2075
100.2.5 Noncompliant Code Example	2075
100.2.6 Compliant Code Example	2076
100.2.7 Notes	2076
100.3 Use a Static Analysis Tool Extensively (SWE03)	2076
100.3.1 Reference	2076
100.3.2 Description	2077
100.3.3 Applicable Vulnerability within ISO TR 24772-2	2077
100.3.4 Applicable Common Weakness Enumeration	2077
100.3.5 Noncompliant Code Example	2077
100.3.6 Compliant Code Example	2077
100.3.7 Notes	2077
100.4 Hide Implementation Artifacts (SWE04)	2077
100.4.1 Reference	2078
100.4.2 Description	2078
100.4.3 Applicable Vulnerability within ISO TR 24772-2	2078

100.4.4	Applicable Common Weakness Enumeration	2078
100.4.5	Noncompliant Code Example	2078
100.4.6	Compliant Code Example	2079
100.4.7	Notes	2079
10	References	2081
X	What's New in Ada 2022	2083
102	Introduction	2087
102.1	References	2087
103	Image attribute for any type	2089
103.1	'Image attribute for a value	2089
103.2	'Image attribute for any type	2089
103.3	References	2090
104	Redefining the 'Image attribute	2091
104.1	What's the Root_Buffer_Type?	2092
104.2	Outdated draft implementation	2092
104.3	References	2092
105	User-Defined Literals	2093
105.1	Turn Ada into JavaScript	2094
105.2	References	2095
106	Advanced Array Aggregates	2097
106.1	Square brackets	2097
106.2	Iterated Component Association	2098
106.3	References	2099
107	Container Aggregates	2101
107.1	References	2105
108	Delta Aggregates	2107
108.1	Delta aggregate for records	2107
108.2	Delta aggregate for arrays	2107
108.3	References	2108
109	Target Name Symbol (@)	2109
109.1	Alternatives	2111
109.2	References	2111
110	Enumeration representation	2113
110.1	Literal positions	2113
110.2	Representation values	2114
110.3	Before Ada 2022	2115
110.4	References	2116
111	Big Numbers	2117
111.1	Big Integers	2117
111.2	Tiny RSA implementation	2117
111.3	Big Reals	2119
111.4	References	2119
112	Interfacing C variadic functions	2121
112.1	References	2122

XI	Introduction to the GNAT Toolchain	2125
113	GNAT Toolchain Basics	2129
113.1	Basic commands	2129
113.2	Compiler warnings	2129
113.2.1	gnatwa switch and warning suppression	2129
113.2.2	Style checking	2132
114	PRbuild	2133
114.1	Basic commands	2133
114.2	Project files	2133
114.2.1	Basic structure	2133
114.2.2	Customization	2134
114.3	Project dependencies	2135
114.3.1	Simple dependency	2135
114.3.2	Dependencies to dynamic libraries	2136
114.4	Configuration pragma files	2137
114.5	Configuration packages	2137
115	GNAT Studio	2139
115.1	Start-up	2139
115.1.1	Windows	2139
115.1.2	Linux	2139
115.2	Creating projects	2139
115.3	Building	2140
115.4	Debugging	2140
115.4.1	Debug information	2140
115.4.2	Improving main application	2141
115.4.3	Debugging the application	2141
115.5	Formal verification	2142
116	GNAT Tools	2145
116.1	gnatchop	2145
116.2	gnatprep	2146
116.3	gnatmem	2147
116.4	gnatdoc	2148
116.5	gnatpp	2150
116.6	gnatstub	2151
XII	Introduction to Ada: Laboratories	2153
117	Imperative language	2157
117.1	Hello World	2157
117.2	Greetings	2157
117.3	Positive Or Negative	2158
117.4	Numbers	2159
118	Subprograms	2161
118.1	Subtract procedure	2161
118.2	Subtract function	2162
118.3	Equality function	2163
118.4	States	2164
118.5	States #2	2165
118.6	States #3	2167
118.7	States #4	2168
119	Modular Programming	2171
119.1	Months	2171

119.2Operations	2172
120Strongly typed language	2175
120.1Colors	2175
120.2Integers	2177
120.3Temperatures	2181
121Records	2185
121.1Directions	2185
121.2Colors	2187
121.3Inventory	2191
122Arrays	2195
122.1Constrained Array	2195
122.2Colors: Lookup-Table	2197
122.3Unconstrained Array	2200
122.4Product info	2203
122.5String_10	2206
122.6List of Names	2208
123More About Types	2213
123.1Aggregate Initialization	2213
123.2Versioning	2215
123.3Simple todo list	2217
123.4Price list	2219
124Privacy	2225
124.1Directions	2225
124.2Limited Strings	2227
124.3Bonus exercise	2231
124.3.1Colors	2231
124.3.2List of Names	2232
124.3.3Price List	2232
125Generics	2233
125.1Display Array	2233
125.2Average of Array of Float	2235
125.3Average of Array of Any Type	2237
125.4Generic list	2240
126Exceptions	2243
126.1Uninitialized Value	2243
126.2Numerical Exception	2244
126.3Re-raising Exceptions	2246
127Tasking	2249
127.1Display Service	2249
127.2Event Manager	2250
127.3Generic Protected Queue	2252
128Design by contracts	2255
128.1Price Range	2255
128.2Pythagorean Theorem: Predicate	2256
128.3Pythagorean Theorem: Precondition	2258
128.4Pythagorean Theorem: Postcondition	2260
128.5Pythagorean Theorem: Type Invariant	2262
128.6Primary Color	2264
129Object-oriented programming	2269
129.1Simple type extension	2269

129.2Online Store	2271
130Standard library: Containers	2277
130.1Simple todo list	2277
130.2List of unique integers	2279
131Standard library: Dates & Times	2283
131.1Holocene calendar	2283
131.2List of events	2284
132Standard library: Strings	2289
132.1Concatenation	2289
132.2List of events	2291
133Standard library: Numerics	2295
133.1Decibel Factor	2295
133.2Root-Mean-Square	2297
133.3Rotation	2299
134Solutions	2305
134.1Imperative Language	2305
134.1.1Hello World	2305
134.1.2Greetings	2305
134.1.3Positive Or Negative	2305
134.1.4Numbers	2306
134.2Subprograms	2307
134.2.1Subtract Procedure	2307
134.2.2Subtract Function	2308
134.2.3Equality function	2309
134.2.4States	2310
134.2.5States #2	2311
134.2.6States #3	2312
134.2.7States #4	2312
134.3Modular Programming	2313
134.3.1Months	2313
134.3.2Operations	2315
134.4Strongly typed language	2317
134.4.1Colors	2317
134.4.2Integers	2318
134.4.3Temperatures	2321
134.5Records	2324
134.5.1Directions	2324
134.5.2Colors	2325
134.5.3Inventory	2328
134.6Arrays	2330
134.6.1Constrained Array	2330
134.6.2Colors: Lookup-Table	2332
134.6.3Unconstrained Array	2334
134.6.4Product info	2336
134.6.5String_10	2339
134.6.6List of Names	2340
134.7More About Types	2343
134.7.1Aggregate Initialization	2343
134.7.2Versioning	2345
134.7.3Simple todo list	2346
134.7.4Price list	2348
134.8Privacy	2350
134.8.1Directions	2350
134.8.2Limited Strings	2352

134.9	Generics	2355
134.9.1	Display Array	2355
134.9.2	Average of Array of Float	2356
134.9.3	Average of Array of Any Type	2358
134.9.4	Generic list	2360
134.10	Exceptions	2362
134.10.1	Uninitialized Value	2362
134.10.2	Numerical Exception	2363
134.10.3	Re-raising Exceptions	2365
134.11	Tasking	2367
134.11.1	Display Service	2367
134.11.2	Event Manager	2368
134.11.3	Generic Protected Queue	2370
134.12	Design by contracts	2372
134.12.1	Price Range	2372
134.12.2	Pythagorean Theorem: Predicate	2373
134.12.3	Pythagorean Theorem: Precondition	2375
134.12.4	Pythagorean Theorem: Postcondition	2377
134.12.5	Pythagorean Theorem: Type Invariant	2378
134.12.6	Primary Colors	2380
134.13	Object-oriented programming	2382
134.13.1	Simple type extension	2382
134.13.2	Online Store	2384
134.14	Standard library: Containers	2387
134.14.1	Simple todo list	2387
134.14.2	List of unique integers	2388
134.15	Standard library: Dates & Times	2390
134.15.1	Holocene calendar	2390
134.15.2	List of events	2391
134.16	Standard library: Strings	2393
134.16.1	Concatenation	2393
134.16.2	List of events	2395
134.17	Standard library: Numerics	2398
134.17.1	Decibel Factor	2398
134.17.2	Root-Mean-Square	2399
134.17.3	Rotation	2402

XIII Bug Free Coding with SPARK Ada 2405

135	Let's Build a Stack	2409
135.1	Background	2409
135.2	Input Format	2411
135.3	Constraints	2412
135.4	Output Format	2412
135.5	Sample Input	2412
135.6	Sample Output	2412

Bibliography 2415

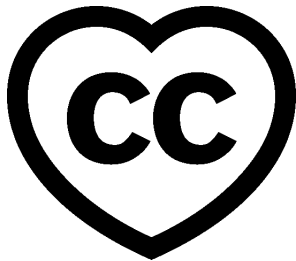
Index 2417

Part I

Introduction to Ada

Copyright © 2018 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)¹



This course will teach you the basics of the Ada programming language and is intended for those who already have a basic understanding of programming techniques. You will learn how to apply those techniques to programming in Ada.

This document was written by Raphaël Amiard and Gustavo A. Hoffmann, with review from Richard Kenner.

Note

The code examples in this course use a 50-column limit, which greatly improves the readability of the code on devices with a small screen size. This constraint, however, leads to an unusual coding style. For instance, instead of calling `Put_Line` in a single line, we have this:

```
Put_Line
(" is in the northeast quadrant");
```

or this:

```
Put_Line (" (X => "
& Integer'Image (P.X)
& ")");
```

Note that typical Ada code uses a limit of at least 79 columns. Therefore, please don't take the coding style from this course as a reference!

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](#)². The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

¹ [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
cba89a34b87c9dfa71533d982d05e6ab/

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

² https://learn.adacore.com/zip/learning-ada_code.zip

INTRODUCTION

1.1 History

In the 1970s the United States Department of Defense (DOD) suffered from an explosion of the number of programming languages, with different projects using different and non-standard dialects or language subsets / supersets. The DOD decided to solve this problem by issuing a request for proposals for a common, modern programming language. The winning proposal was one submitted by Jean Ichbiah from CII Honeywell-Bull.

The first Ada standard was issued in 1983; it was subsequently revised and enhanced in 1995, 2005 and 2012, with each revision bringing useful new features.

This tutorial will focus on Ada 2012 as a whole, rather than teaching different versions of the language.

1.2 Ada today

Today, Ada is heavily used in embedded real-time systems, many of which are safety critical. While Ada is and can be used as a general-purpose language, it will really shine in low-level applications:

- Embedded systems with low memory requirements (no garbage collector allowed).
- Direct interfacing with hardware.
- Soft or hard real-time systems.
- Low-level systems programming.

Specific domains seeing Ada usage include Aerospace & Defense, civil aviation, rail, and many others. These applications require a high degree of safety: a software defect is not just an annoyance, but may have severe consequences. Ada provides safety features that detect defects at an early stage — usually at compilation time or using static analysis tools. Ada can also be used to create applications in a variety of other areas, such as:

- Video game programming³
- Real-time audio⁴
- Kernel modules⁵

This is a non-comprehensive list that hopefully sheds light on which kind of programming Ada is good at.

³ <https://github.com/AdaDoom3/AdaDoom3>

⁴ <http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications>

⁵ <http://www.nihamkin.com/tag/kernel.html>

In terms of modern languages, the closest in terms of targets and level of abstraction are probably C++⁶ and Rust⁷.

1.3 Philosophy

Ada's philosophy is different from most other languages. Underlying Ada's design are principles that include the following:

- Readability is more important than conciseness. Syntactically this shows through the fact that keywords are preferred to symbols, that no keyword is an abbreviation, etc.
- Very strong typing. It is very easy to introduce new types in Ada, with the benefit of preventing data usage errors.
 - It is similar to many functional languages in that regard, except that the programmer has to be much more explicit about typing in Ada, because there is almost no type inference.
- Explicit is better than implicit. Although this is a Python⁸ commandment, Ada takes it way further than any language we know of:
 - There is mostly no structural typing, and most types need to be explicitly named by the programmer.
 - As previously said, there is mostly no type inference.
 - Semantics are very well defined, and undefined behavior is limited to an absolute minimum.
 - The programmer can generally give a *lot* of information about what their program means to the compiler (and other programmers). This allows the compiler to be extremely helpful (read: strict) with the programmer.

During this course, we will explain the individual language features that are building blocks for that philosophy.

1.4 SPARK

While this class is solely about the Ada language, it is worth mentioning that another language, extremely close to and interoperable with Ada, exists: the SPARK language.

SPARK is a subset of Ada, designed so that the code written in SPARK is amenable to automatic proof. This provides a level of assurance with regard to the correctness of your code that is much higher than with a regular programming language.

There is a dedicated [course for the SPARK language](#) (page 1411) but keep in mind that every time we speak about the specification power of Ada during this course, it is power that you can leverage in SPARK to help proving the correctness of program properties ranging from absence of run-time errors to compliance with formally specified functional requirements.

⁶ <https://en.wikipedia.org/wiki/C%2B%2B>

⁷ <https://www.rust-lang.org/en-US/>

⁸ <https://www.python.org>

IMPERATIVE LANGUAGE

Ada is a multi-paradigm language with support for object orientation and some elements of functional programming, but its core is a simple, coherent procedural/imperative language akin to C or Pascal.

In other languages

One important distinction between Ada and a language like C is that statements and expressions are very clearly distinguished. In Ada, if you try to use an expression where a statement is required then your program will fail to compile. This rule supports a useful stylistic principle: expressions are intended to deliver values, not to have side effects. It can also prevent some programming errors, such as mistakenly using the equality operator `=` instead of the assignment operation `:=` in an assignment statement.

2.1 Hello world

Here's a very simple imperative Ada program:

Listing 1: greet.adb

```
1 with Ada.Text_IO;
2
3 procedure Greet is
4 begin
5     -- Print "Hello, World!" to the screen
6     Ada.Text_IO.Put_Line ("Hello, World!");
7 end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet
MD5: cba89a34b87c9dfa71533d982d05e6ab

Runtime output

Hello, World!

which we'll assume is in the source file `greet.adb`.

There are several noteworthy things in the above program:

- A subprogram in Ada can be either a procedure or a function. A procedure, as illustrated above, does not return a value when called.
- **with** is used to reference external modules that are needed in the procedure. This is similar to `import` in various languages or roughly similar to `#include` in C and C++. We'll see later how they work in detail. Here, we are requesting a standard library

module, the `Ada.Text_IO` package, which contains a procedure to print text on the screen: `Put_Line`.

- `Greet` is a procedure, and the main entry point for our first program. Unlike in C or C++, it can be named anything you prefer. The builder will determine the entry point. In our simple example, **gprbuild**, GNAT's builder, will use the file you passed as parameter.
- `Put_Line` is a procedure, just like `Greet`, except it is declared in the `Ada.Text_IO` module. It is the Ada equivalent of C's `printf`.
- Comments start with `--` and go to the end of the line. There is no multi-line comment syntax, that is, it is not possible to start a comment in one line and continue it in the next line. The only way to create multiple lines of comments in Ada is by using `--` on each line. For example:

```
-- We start a comment in this line...
-- and we continue on the second line...
```

In other languages

Procedures are similar to functions in C or C++ that return **void**. We'll see later how to declare functions in Ada.

Here is a minor variant of the "Hello, World" example:

Listing 2: `greet.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4 begin
5     -- Print "Hello, World!" to the screen
6     Put_Line ("Hello, World!");
7 end Greet;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Imperative_Language.Greet_2`
MD5: `a58a1193207df44aa6edaa4fe1c14280`

Runtime output

Hello, World!

This version utilizes an Ada feature known as a **use** clause, which has the form **use package-name**. As illustrated by the call on `Put_Line`, the effect is that entities from the named package can be referenced directly, without the *package-name*. prefix.

2.2 Imperative language - If/Then/Else

This section describes Ada's **if** statement and introduces several other fundamental language facilities including integer I/O, data declarations, and subprogram parameter modes.

Ada's **if** statement is pretty unsurprising in form and function:

Listing 3: check_positive.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
5      N : Integer;
6  begin
7      -- Put a String
8      Put ("Enter an integer value: ");
9
10     -- Read in an integer value
11     Get (N);
12
13     if N > 0 then
14         -- Put an Integer
15         Put (N);
16         Put_Line (" is a positive number");
17     end if;
18 end Check_Positive;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Positive
MD5: 2e8b4b2f3f258fd9e02c2d65846af101

The **if** statement minimally consists of the reserved word **if**, a condition (which must be a Boolean value), the reserved word **then** and a non-empty sequence of statements (the **then** part) which is executed if the condition evaluates to True, and a terminating **end if**.

This example declares an integer variable N, prompts the user for an integer, checks if the value is positive and, if so, displays the integer's value followed by the string " is a positive number". If the value is not positive, the procedure does not display any output.

The type Integer is a predefined signed type, and its range depends on the computer architecture. On typical current processors Integer is 32-bit signed.

The example illustrates some of the basic functionality for integer input-output. The relevant subprograms are in the predefined package Ada.Integer_Text_IO and include the Get procedure (which reads in a number from the keyboard) and the Put procedure (which displays an integer value).

Here's a slight variation on the example, which illustrates an **if** statement with an **else** part:

Listing 4: check_positive.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
5      N : Integer;
6  begin
7      -- Put a String
8      Put ("Enter an integer value: ");
9
10     -- Reads in an integer value
11     Get (N);
12
13     -- Put an Integer
14     Put (N);
15

```

(continues on next page)

(continued from previous page)

```
16  if N > 0 then
17      Put_Line (" is a positive number");
18  else
19      Put_Line (" is not a positive number");
20  end if;
21  end Check_Positive;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Positive_2
MD5: 28fca0d7840d06d478e5933e8182d1db

In this example, if the input value is not positive then the program displays the value followed by the String " is not a positive number".

Our final variation illustrates an **if** statement with **elsif** sections:

Listing 5: check_direction.adb

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4  procedure Check_Direction is
5      N : Integer;
6  begin
7      Put ("Enter an integer value: ");
8      Get (N);
9      Put (N);
10
11     if N = 0 or N = 360 then
12         Put_Line (" is due north");
13     elsif N in 1 .. 89 then
14         Put_Line (" is in the northeast quadrant");
15     elsif N = 90 then
16         Put_Line (" is due east");
17     elsif N in 91 .. 179 then
18         Put_Line (" is in the southeast quadrant");
19     elsif N = 180 then
20         Put_Line (" is due south");
21     elsif N in 181 .. 269 then
22         Put_Line (" is in the southwest quadrant");
23     elsif N = 270 then
24         Put_Line (" is due west");
25     elsif N in 271 .. 359 then
26         Put_Line (" is in the northwest quadrant");
27     else
28         Put_Line (" is not in the range 0..360");
29     end if;
30 end Check_Direction;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Direction
MD5: 7759d30c9bb0bfb88efdf12128f9c382

This example expects the user to input an integer between 0 and 360 inclusive, and displays which quadrant or axis the value corresponds to. The **in** operator in Ada tests whether a scalar value is within a specified range and returns a Boolean result. The effect of the program should be self-explanatory; later we'll see an alternative and more efficient style to accomplish the same effect, through a **case** statement.

Ada's **elsif** keyword differs from C or C++, where nested **else** .. **if** blocks would be

used instead. And another difference is the presence of the **end if** in Ada, which avoids the problem known as the "dangling else".

2.3 Imperative language - Loops

Ada has three ways of specifying loops. They differ from the C / Java / Javascript for-loop, however, with simpler syntax and semantics in line with Ada's philosophy.

2.3.1 For loops

The first kind of loop is the **for** loop, which allows iteration through a discrete range.

Listing 6: greet_5a.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5a is
4 begin
5   for I in 1 .. 5 loop
6     -- Put_Line is a procedure call
7     Put_Line ("Hello, World!"
8               & Integer'Image (I));
9     --      ^ Procedure parameter
10  end loop;
11 end Greet_5a;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5a
MD5: 7f588b67947126f789333adfaaf1b638

Runtime output

```

Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

A few things to note:

- **1 .. 5** is a discrete range, from **1** to **5** inclusive.
- The loop parameter **I** (the name is arbitrary) in the body of the loop has a value within this range.
- **I** is local to the loop, so you cannot refer to **I** outside the loop.
- Although the value of **I** is incremented at each iteration, from the program's perspective it is constant. An attempt to modify its value is illegal; the compiler would reject the program.
- **Integer'Image** is a function that takes an Integer and converts it to a **String**. It is an example of a language construct known as an *attribute*, indicated by the **'** syntax, which will be covered in more detail later.
- The **&** symbol is the concatenation operator for String values
- The **end loop** marks the end of the loop

The "step" of the loop is limited to 1 (forward direction) and -1 (backward). To iterate backwards over a range, use the **reverse** keyword:

Listing 7: greet_5a_reverse.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5a_Reverse is
4 begin
5     for I in reverse 1 .. 5 loop
6         Put_Line ("Hello, World!"
7                 & Integer'Image (I));
8     end loop;
9 end Greet_5a_Reverse;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5a_Reverse
MD5: a0d5dcfc471fb1a107477c934fa527c2

Runtime output

```
Hello, World! 5
Hello, World! 4
Hello, World! 3
Hello, World! 2
Hello, World! 1
```

The bounds of a **for** loop may be computed at run-time; they are evaluated once, before the loop body is executed. If the value of the upper bound is less than the value of the lower bound, then the loop is not executed at all. This is the case also for **reverse** loops. Thus no output is produced in the following example:

Listing 8: greet_no_op.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_No_Op is
4 begin
5     for I in reverse 5 .. 1 loop
6         Put_Line ("Hello, World!"
7                 & Integer'Image (I));
8     end loop;
9 end Greet_No_Op;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_No_Op
MD5: 5070693fb0324d3e4e43a8c8c4f046e1

Build output

```
greet_no_op.adb:5:23: warning: loop range is null, loop will not execute [enabled_
↳by default]
```

The **for** loop is more general than what we illustrated here; more on that later.

2.3.2 Bare loops

The simplest loop in Ada is the bare loop, which forms the foundation of the other kinds of Ada loops.

Listing 9: greet_5b.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5b is
4   -- Variable declaration:
5   I : Integer := 1;
6   -- ^ Type
7   -- ^ Initial value
8 begin
9   loop
10    Put_Line ("Hello, World!"
11              & Integer'Image (I));
12
13    -- Exit statement:
14    exit when I = 5;
15    -- ^ Boolean condition
16
17    -- Assignment:
18    I := I + 1;
19    -- There is no I++ short form to
20    -- increment a variable
21  end loop;
22 end Greet_5b;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5b
MD5: 5b218a64a07f64bd97774b574883c44a

Runtime output

```

Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5

```

This example has the same effect as Greet_5a shown earlier.

It illustrates several concepts:

- We have declared a variable named `I` between the `is` and the `begin`. This constitutes a *declarative region*. Ada clearly separates the declarative region from the statement part of a subprogram. A declaration can appear in a declarative region but is not allowed as a statement.
- The bare loop statement is introduced by the keyword `loop` on its own and, like every kind of loop statement, is terminated by the combination of keywords `end loop`. On its own, it is an infinite loop. You can break out of it with an `exit` statement.
- The syntax for assignment is `:=`, and the one for equality is `=`. There is no way to confuse them, because as previously noted, in Ada, statements and expressions are distinct, and expressions are not valid statements.

2.3.3 While loops

The last kind of loop in Ada is the `while` loop.

Listing 10: greet_5c.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5c is
4   I : Integer := 1;
5 begin
6   -- Condition must be a Boolean value
7   -- (no Integers).
8   -- Operator "<=" returns a Boolean
9   while I <= 5 loop
10    Put_Line ("Hello, World!"
11              & Integer'Image (I));
12
13    I := I + 1;
14  end loop;
15 end Greet_5c;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_5c
MD5: 5d1d099477795b226db43736c2810274

Runtime output

```

Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5

```

The condition is evaluated before each iteration. If the result is false, then the loop is terminated.

This program has the same effect as the previous examples.

** In other languages**

Note that Ada has different semantics than C-based languages with respect to the condition in a while loop. In Ada the condition has to be a Boolean value or the compiler will reject the program; the condition is not an integer that is treated as either **True** or **False** depending on whether it is non-zero or zero.

2.4 Imperative language - Case statement

Ada's **case** statement is similar to the C and C++ **switch** statement, but with some important differences.

Here's an example, a variation of a program that was shown earlier with an **if** statement:

Listing 11: check_direction.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4 procedure Check_Direction is
5   N : Integer;
6 begin

```

(continues on next page)

(continued from previous page)

```

7  loop
8      Put ("Enter an integer value: ");
9      Get (N);
10     Put (N);
11
12     case N is
13         when 0 | 360 =>
14             Put_Line
15                 (" is due north");
16         when 1 .. 89 =>
17             Put_Line
18                 (" is in the northeast quadrant");
19         when 90 =>
20             Put_Line
21                 (" is due east");
22         when 91 .. 179 =>
23             Put_Line
24                 (" is in the southeast quadrant");
25         when 180 =>
26             Put_Line
27                 (" is due south");
28         when 181 .. 269 =>
29             Put_Line
30                 (" is in the southwest quadrant");
31         when 270 =>
32             Put_Line
33                 (" is due west");
34         when 271 .. 359 =>
35             Put_Line
36                 (" is in the northwest quadrant");
37         when others =>
38             Put_Line
39                 (" Au revoir");
40             exit;
41     end case;
42 end loop;
43 end Check_Direction;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Direction_2
MD5: 1c758b76a2c3991cb4e2a0cf5e172ac3

This program repeatedly prompts for an integer value and then, if the value is in the range `0 .. 360`, displays the associated quadrant or axis. If the value is an Integer outside this range, the loop (and the program) terminate after outputting a farewell message.

The effect of the case statement is similar to the if statement in an earlier example, but the case statement can be more efficient because it does not involve multiple range tests.

Notable points about Ada's case statement:

- The case expression (here the variable `N`) must be of a discrete type, i.e. either an integer type or an enumeration type. Discrete types will be covered in more detail later [discrete types](#) (page 47).
- Every possible value for the case expression needs to be covered by a unique branch of the case statement. This will be checked at compile time.
- A branch can specify a single value, such as `0`; a range of values, such as `1 .. 89`; or any combination of the two (separated by a `|`).
- As a special case, an optional final branch can specify `others`, which covers all values

not included in the earlier branches.

- Execution consists of the evaluation of the case expression and then a transfer of control to the statement sequence in the unique branch that covers that value.
- When execution of the statements in the selected branch has completed, control resumes after the **end case**. Unlike C, execution does not fall through to the next branch. So Ada doesn't need (and doesn't have) a **break** statement.

2.5 Imperative language - Declarative regions

As mentioned earlier, Ada draws a clear syntactic separation between declarations, which introduce names for entities that will be used in the program, and statements, which perform the processing. The areas in the program where declarations may appear are known as declarative regions.

In any subprogram, the section between the **is** and the **begin** is a declarative region. You can have variables, constants, types, inner subprograms, and other entities there.

We've briefly mentioned variable declarations in previous subsection. Let's look at a simple example, where we declare an integer variable X in the declarative region and perform an initialization and an addition on it:

Listing 12: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   X : Integer;
5 begin
6   X := 0;
7   Put_Line ("The initial value of X is "
8             & Integer'Image (X));
9
10  Put_Line ("Performing operation on X...");
11  X := X + 1;
12
13  Put_Line ("The value of X now is "
14            & Integer'Image (X));
15 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Variable_Declaration
MD5: cbb08d5e382fbfcc28e986bea80cd253

Runtime output

```
The initial value of X is 0
Performing operation on X...
The value of X now is 1
```

Let's look at an example of a nested procedure:

Listing 13: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   procedure Nested is
5     begin
```

(continues on next page)

(continued from previous page)

```

6      Put_Line ("Hello World");
7  end Nested;
8  begin
9      Nested;
10     -- Call to Nested
11 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Nested_Procedure
MD5: 2e7fb267e31232196065febd5e35e6ef

Runtime output

Hello World

A declaration cannot appear as a statement. If you need to declare a local variable amidst the statements, you can introduce a new declarative region with a block statement:

Listing 14: greet.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4  begin
5      loop
6          Put_Line ("Please enter your name: ");
7
8          declare
9              Name : String := Get_Line;
10             --           ^ Call to the
11             --           Get_Line function
12         begin
13             exit when Name = "";
14             Put_Line ("Hi " & Name & "!");
15         end;
16
17         -- Name is undefined here
18     end loop;
19
20     Put_Line ("Bye!");
21 end Greet;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Greet_6
MD5: a9c0c14a1b3e2ebe07cd88f442787e3a

⚠ Attention

The `Get_Line` function allows you to receive input from the user, and get the result as a string. It is more or less equivalent to the `scanf` C function.

It returns a **String**, which, as we will see later, is an *Unconstrained array type* (page 78). For now we simply note that, if you wish to declare a **String** variable and do not know its size in advance, then you need to initialize the variable during its declaration.

2.6 Imperative language - conditional expressions

Ada 2012 introduced an expression analog for conditional statements (**if** and **case**).

2.6.1 If expressions

Here's an alternative version of an example we saw earlier; the **if** statement has been replaced by an **if** expression:

Listing 15: check_positive.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
5      N : Integer;
6  begin
7      Put ("Enter an integer value: ");
8      Get (N);
9      Put (N);
10
11     declare
12         S : constant String :=
13             (if N > 0
14              then " is a positive number"
15              else " is not a positive number");
16     begin
17         Put_Line (S);
18     end;
19 end Check_Positive;
```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Check_Positive
MD5: 01f23463b14774f750dbb21f6c65ea09

The **if** expression evaluates to one of the two Strings depending on N, and assigns that value to the local variable S.

Ada's **if** expressions are similar to **if** statements. However, there are a few differences that stem from the fact that it is an expression:

- All branches' expressions must be of the same type
- It *must* be surrounded by parentheses if the surrounding expression does not already contain them
- An **else** branch is mandatory unless the expression following **then** has a Boolean value. In that case an **else** branch is optional and, if not present, defaults to **else True**.

Here's another example:

Listing 16: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4  begin
5      for I in 1 .. 10 loop
6          Put_Line (if I mod 2 = 0
7                      then "Even"
```

(continues on next page)

(continued from previous page)

```

8         else "Odd");
9     end loop;
10 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Even_Odd
MD5: c89c3233ab8822c828f7a7bba8fd3f1c

Runtime output

```

Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even

```

This program produces 10 lines of output, alternating between "Odd" and "Even".

2.6.2 Case expressions

Analogous to **if** expressions, Ada also has **case** expressions. They work just as you would expect.

Listing 17: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     for I in 1 .. 10 loop
6         Put_Line
7             (case I is
8                 when 1 | 3 | 5 | 7 | 9 => "Odd",
9                 when 2 | 4 | 6 | 8 | 10 => "Even");
10    end loop;
11 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Imperative_Language.Case_Expression
MD5: 6ce40efc987c2665960b1f08d30d780d

Runtime output

```

Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even

```


This program has the same effect as the preceding example.

The syntax differs from **case** statements, with branches separated by commas.

SUBPROGRAMS

3.1 Subprograms

So far, we have used procedures, mostly to have a main body of code to execute. Procedures are one kind of *subprogram*.

There are two kinds of subprograms in Ada, *functions* and *procedures*. The distinction between the two is that a function returns a value, and a procedure does not.

This example shows the declaration and definition of a function:

Listing 18: increment.ads

```
1 function Increment (I : Integer) return Integer;
```

Listing 19: increment.adb

```
1 -- We declare (but don't define) a function with
2 -- one parameter, returning an integer value
3
4 function Increment (I : Integer) return Integer is
5     -- We define the Increment function
6 begin
7     return I + 1;
8 end Increment;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Subprograms.Increment
MD5: 582fe283730a130cec071c455a0ce3d4
```

Subprograms in Ada can, of course, have parameters. One syntactically important note is that a subprogram which has no parameters does not have a parameter section at all, for example:

```
procedure Proc;

function Func return Integer;
```

Here's another variation on the previous example:

Listing 20: increment_by.ads

```
1 function Increment_By
2     (I : Integer := 0;
3      Incr : Integer := 1) return Integer;
4 -- ^ Default value for parameters
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Increment_By
MD5: 5728b915789beee0b5546ea7b36a1cc2

In this example, we see that parameters can have default values. When calling the subprogram, you can then omit parameters if they have a default value. Unlike C/C++, a call to a subprogram without parameters does not include parentheses.

This is the implementation of the function above:

Listing 21: increment_by.adb

```
1 function Increment_By
2   (I      : Integer := 0;
3    Incr   : Integer := 1) return Integer is
4 begin
5   return I + Incr;
6 end Increment_By;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Increment_By
MD5: 07c85e5c1272ea396bf4dbc0cefcfdce7

In the GNAT toolchain

The Ada standard doesn't mandate in which file the specification or the implementation of a subprogram must be stored. In other words, the standard doesn't require a specific file structure or specific file name extensions. For example, we could save both the specification and the implementation of the Increment function above in a file called `increment.txt`. (We could even store the entire source code of a system in a single file.) From the standard's perspective, this would be completely acceptable.

The GNAT toolchain, however, requires the following file naming scheme:

- files with the `.ads` extension contain the specification, while
- files with the `.adb` extension contain the implementation.

Therefore, in the GNAT toolchain, the specification of the Increment function must be stored in the `increment.ads` file, while its implementation must be stored in the `increment.adb` file. This rule always applies to packages, which we discuss *later* (page 35). (Note, however, that it's possible to circumvent this rule.) For more details, you may refer to the [Introduction to GNAT Toolchain](#) (page 2127) course or the [GPRbuild User's Guide](#)⁹.

3.1.1 Subprogram calls

We can then call our subprogram this way:

Listing 22: show_increment.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Increment_By;
3
4 procedure Show_Increment is
5   A, B, C : Integer;
6 begin
7   C := Increment_By;
```

(continues on next page)

⁹ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

(continued from previous page)

```

8      --      ^ Parameterless call,
9      --      value of I is 0
10     --      and Incr is 1
11
12     Put_Line ("Using defaults for Increment_By is "
13              & Integer'Image (C));
14
15     A := 10;
16     B := 3;
17     C := Increment_By (A, B);
18     --      ^ Regular parameter passing
19
20     Put_Line ("Increment of "
21              & Integer'Image (A)
22              & " with "
23              & Integer'Image (B)
24              & " is "
25              & Integer'Image (C));
26
27     A := 20;
28     B := 5;
29     C := Increment_By (I => A,
30                      Incr => B);
31     --      ^ Named parameter passing
32
33     Put_Line ("Increment of "
34              & Integer'Image (A)
35              & " with "
36              & Integer'Image (B)
37              & " is "
38              & Integer'Image (C));
39 end Show_Increment;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Increment_By
MD5: dcb501c8c6815b03c6841fc8b80d6911

Runtime output

```

Using defaults for Increment_By is 1
Increment of 10 with 3 is 13
Increment of 20 with 5 is 25

```

Ada allows you to name the parameters when you pass them, whether they have a default or not. There are some rules:

- Positional parameters come first.
- A positional parameter cannot follow a named parameter.

As a convention, people usually name parameters at the call site if the function's corresponding parameters has a default value. However, it is also perfectly acceptable to name every parameter if it makes the code clearer.

3.1.2 Nested subprograms

As briefly mentioned earlier, Ada allows you to declare one subprogram inside another.

This is useful for two reasons:

- It lets you organize your programs in a cleaner fashion. If you need a subprogram only as a "helper" for another subprogram, then the principle of localization indicates that the helper subprogram should be declared nested.
- It allows you to share state easily in a controlled fashion, because the nested subprograms have access to the parameters, as well as any local variables, declared in the outer scope.

For the previous example, we can move the duplicated code (call to `Put_Line`) to a separate procedure. This is a shortened version with the nested `Display_Result` procedure.

Listing 23: `show_increment.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Increment_By;
3
4 procedure Show_Increment is
5   A, B, C : Integer;
6
7   procedure Display_Result is
8   begin
9     Put_Line ("Increment of "
10              & Integer'Image (A)
11              & " with "
12              & Integer'Image (B)
13              & " is "
14              & Integer'Image (C));
15   end Display_Result;
16
17 begin
18   A := 10;
19   B := 3;
20   C := Increment_By (A, B);
21   Display_Result;
22   A := 20;
23   B := 5;
24   C := Increment_By (A, B);
25   Display_Result;
26 end Show_Increment;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Increment_By
MD5: 23ec8ae3080c042123a9e82ee6b3d9e3

Runtime output

```
Increment of 10 with 3 is 13
Increment of 20 with 5 is 25
```

3.1.3 Function calls

An important feature of function calls in Ada is that the return value at a call cannot be ignored; that is, a function call cannot be used as a statement.

If you want to call a function and do not need its result, you will still need to explicitly store it in a local variable.

Listing 24: `quadruple.ads`

```
1 function Quadruple (I : Integer)
2   return Integer;
```

Listing 25: quadruple.adb

```

1 function Quadruple (I : Integer)
2     return Integer is
3
4     function Double (I : Integer)
5         return Integer is
6
7     begin
8         return I * 2;
9     end Double;
10
11 Res : Integer;
12 begin
13     Double (Double (I));
14     -- ERROR: cannot use call to function
15     --         "Double" as a statement
16
17     Res := Double (Double (I));
18     -- OK: return value of Double is
19     --         assigned to Res
20
21 return Res;
end Quadruple;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Quadruple
MD5: fd525bde1a883d4a3794730695af6469

Build output

```

quadruple.adb:12:04: error: cannot use call to function "Double" as a statement
quadruple.adb:12:04: error: return value of a function call cannot be ignored
gprbuild: *** compilation phase failed

```

A statement such as `Double (Double (I));` is wrong because we're not assigning the return value to a variable — we can correct this statement by writing `Res := Double (Double (I));`.

In the GNAT toolchain

In GNAT, with all warnings activated, it becomes even harder to ignore the result of a function, because unused variables will be flagged. For example, this code would not be valid:

```

function Read_Int
(Stream : Network_Stream;
 Result : out Integer) return Boolean;

procedure Main is
    Stream : Network_Stream := Get_Stream;
    My_Int : Integer;

    -- Warning: in the line below, B is
    --         never read.
    B : Boolean := Read_Int (Stream, My_Int);
begin
    null;
end Main;

```

You then have two solutions to silence this warning:

- Either annotate the variable with **pragma Unreferenced**, e.g.:

```
B : Boolean := Read_Int (Stream, My_Int);
pragma Unreferenced (B);
```

- Or give the variable a name that contains any of the strings discard dummy ignore junk unused (case insensitive)

3.2 Parameter modes

So far we have seen that Ada is a safety-focused language. There are many ways this is realized, but two important points are:

- Ada makes the user specify as much as possible about the behavior expected for the program, so that the compiler can warn or reject if there is an inconsistency.
- Ada provides a variety of techniques for achieving the generality and flexibility of pointers and dynamic memory management, but without the latter's drawbacks (such as memory leakage and dangling references).

Parameter modes are a feature that helps achieve the two design goals above. A subprogram parameter can be specified with a mode, which is one of the following:

in	Parameter can only be read, not written
out	Parameter can be written to, then read
in out	Parameter can be both read and written

The default mode for parameters is **in**; so far, most of the examples have been using **in** parameters.

Historically

Functions and procedures were originally more different in philosophy. Before Ada 2012, functions could only take **in** parameters.

3.3 Subprogram calls

3.3.1 In parameters

The first mode for parameters is the one we have been implicitly using so far. Parameters passed using this mode cannot be modified, so that the following program will cause an error:

Listing 26: swap.adb

```
1 procedure Swap (A, B : Integer) is
2   Tmp : Integer;
3 begin
4   Tmp := A;
5
6   -- Error: assignment to "in" mode
7   --       parameter not allowed
8   A := B;
9
10  -- Error: assignment to "in" mode
```

(continues on next page)

(continued from previous page)

```

11      --           parameter not allowed
12      B := Tmp;
13  end Swap;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Swap
MD5: 478ac23f878934aae820e4b9c056d939

Build output

```

swap.adb:8:04: error: assignment to "in" mode parameter not allowed
swap.adb:12:04: error: assignment to "in" mode parameter not allowed
gprbuild: *** compilation phase failed

```

The fact that **in** is the default mode is very important. It means that a parameter will not be modified unless you explicitly specify a mode in which modification is allowed.

3.3.2 In out parameters

To correct our code above, we can use an **in out** parameter.

Listing 27: in_out_params.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure In_Out_Params is
4      procedure Swap (A, B : in out Integer) is
5          Tmp : Integer;
6          begin
7              Tmp := A;
8              A := B;
9              B := Tmp;
10         end Swap;
11
12         A : Integer := 12;
13         B : Integer := 44;
14     begin
15         Swap (A, B);
16
17         -- Prints 44
18         Put_Line (Integer'Image (A));
19     end In_Out_Params;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.In_Out_Params
MD5: 319358e479449c115cf2b3cbb4ff3a6b

Runtime output

44

An **in out** parameter will allow read and write access to the object passed as parameter, so in the example above, we can see that A is modified after the call to Swap.

Attention

While **in out** parameters look a bit like references in C++, or regular parameters in Java that are passed by-reference, the Ada language standard does not mandate "by reference" passing for in out parameters except for certain categories of types as will be explained later.

In general, it is better to think of modes as higher level than by-value versus by-reference semantics. For the compiler, it means that an array passed as an **in** parameter might be passed by reference, because it is more efficient (which does not change anything for the user since the parameter is not assignable). However, a parameter of a discrete type will always be passed by copy, regardless of its mode (which is more efficient on most architectures).

3.3.3 Out parameters

The **out** mode applies when the subprogram needs to write to a parameter that might be uninitialized at the point of call. Reading the value of an **out** parameter is permitted, but it should only be done after the subprogram has assigned a value to the parameter. Out parameters behave a bit like return values for functions. When the subprogram returns, the actual parameter (a variable) will have the value of the out parameter at the point of return.

In other languages

Ada doesn't have a tuple construct and does not allow returning multiple values from a subprogram (except by declaring a full-fledged record type). Hence, a way to return multiple values from a subprogram is to use **out** parameters.

For example, a procedure reading integers from the network could have one of the following specifications:

```

procedure Read_Int
  (Stream : Network_Stream;
   Success : out Boolean;
   Result  : out Integer);

function Read_Int
  (Stream : Network_Stream;
   Result : out Integer) return Boolean;

```

While reading an out variable before writing to it should, ideally, trigger an error, imposing that as a rule would cause either inefficient run-time checks or complex compile-time rules. So from the user's perspective an out parameter acts like an uninitialized variable when the subprogram is invoked.

In the GNAT toolchain

GNAT will detect simple cases of incorrect use of out parameters. For example, the compiler will emit a warning for the following program:

Listing 28: outp.adb

```

1 procedure Outp is
2   procedure Foo (A : out Integer) is
3     B : Integer := A;
4     --           ^ Warning on reference
5     --           to uninitialized A
6   begin

```

```

7     A := B;
8     end Foo;
9     begin
10    null;
11    end Outp;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Out_Params
MD5: 36bdb4e541297d7fb0b075816cb6e73a

Build output

outp.adb:3:22: warning: "A" may be referenced before it has a value [enabled by default]

3.3.4 Forward declaration of subprograms

As we saw earlier, a subprogram can be declared without being fully defined, This is possible in general, and can be useful if you need subprograms to be mutually recursive, as in the example below:

Listing 29: mutually_recursive_subprograms.adb

```

1  procedure Mutually_Recursive_Subprograms is
2      procedure Compute_A (V : Natural);
3      -- Forward declaration of Compute_A
4
5      procedure Compute_B (V : Natural) is
6      begin
7          if V > 5 then
8              Compute_A (V - 1);
9              -- Call to Compute_A
10         end if;
11     end Compute_B;
12
13     procedure Compute_A (V : Natural) is
14     begin
15         if V > 2 then
16             Compute_B (V - 1);
17             -- Call to Compute_B
18         end if;
19     end Compute_A;
20 begin
21     Compute_A (15);
22 end Mutually_Recursive_Subprograms;

```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Mutually_Recursive_Subprograms
MD5: 5ee030cdecc6c4aea8916cbb763e8526

3.4 Renaming

Subprograms can be renamed by using the **renames** keyword and declaring a new name for a subprogram:

```

procedure New_Proc renames Original_Proc;

```

This can be useful, for example, to improve the readability of your application when you're using code from external sources that cannot be changed in your system. Let's look at an example:

Listing 30: a_procedure_with_very_long_name_that_cannot_be_changed.ads

```
1 procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
2   (A_Message : String);
```

Listing 31: a_procedure_with_very_long_name_that_cannot_be_changed.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
4   (A_Message : String) is
5   begin
6     Put_Line (A_Message);
7   end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Proc_Renaming
MD5: 6d4952e9dee8ef69a9e3c3e185c635f1

As the wording in the name of procedure above implies, we cannot change its name. We can, however, rename it to something like `Show` in our test application and use this shorter name. Note that we also have to declare all parameters of the original subprogram — we may rename them, too, in the declaration. For example:

Listing 32: show_renaming.adb

```
1 with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
2
3 procedure Show_Renaming is
4
5     procedure Show (S : String) renames
6       A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
7
8   begin
9     Show ("Hello World!");
10  end Show_Renaming;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Proc_Renaming
MD5: 5b3b550f8a1cbeb7d9cfd3673f6d42b3

Runtime output

Hello World!

Note that the original name (`A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed`) is still visible after the declaration of the `Show` procedure.

We may also rename subprograms from the standard library. For example, we may rename `Integer'Image` to `Img`:

Listing 33: show_image_renaming.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Image_Renaming is
```

(continues on next page)

(continued from previous page)

```
4
5  function Img (I : Integer) return String
6      renames Integer'Image;
7
8  begin
9      Put_Line (Img (2));
10     Put_Line (Img (3));
11 end Show_Image_Renaming;
```

Code block metadata

Project: Courses.Intro_To_Ada.Subprograms.Integer_Image_Renaming
MD5: 9843b9d5967679c4fe8bd83a5213829f

Runtime output

```
2
3
```

Renaming also allows us to introduce default expressions that were not available in the original declaration. For example, we may specify `"Hello World!"` as the default for the `String` parameter of the `Show` procedure:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming_Defaults is

    procedure Show (S : String := "Hello World!")
        renames
            A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
    Show;
end Show_Renaming_Defaults;
```


MODULAR PROGRAMMING

So far, our examples have been simple standalone subprograms. Ada is helpful in that regard, since it allows arbitrary declarations in a declarative part. We were thus able to declare our types and variables in the bodies of main procedures.

However, it is easy to see that this is not going to scale up for real-world applications. We need a better way to structure our programs into modular and distinct units.

Ada encourages the separation of programs into multiple packages and sub-packages, providing many tools to a programmer on a quest for a perfectly organized code-base.

4.1 Packages

Here is an example of a package declaration in Ada:

Listing 34: week.ads

```
1 package Week is
2
3     Mon : constant String := "Monday";
4     Tue : constant String := "Tuesday";
5     Wed : constant String := "Wednesday";
6     Thu : constant String := "Thursday";
7     Fri : constant String := "Friday";
8     Sat : constant String := "Saturday";
9     Sun : constant String := "Sunday";
10
11 end Week;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Week
MD5: 0fa033dc8fe2b9741483de273354e7ee

And here is how you use it:

Listing 35: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week;
3 -- References the Week package, and
4 -- adds a dependency from Main to Week
5
6 procedure Main is
7 begin
8     Put_Line ("First day of the week is "
9              & Week.Mon);
10 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Week
MD5: 03e17a75620de6a397b1d3c5a3e22f6a

Runtime output

First day of the week is Monday

Packages let you make your code modular, separating your programs into semantically significant units. Additionally the separation of a package's specification from its body (which we will see below) can reduce compilation time.

While the **with** clause indicates a dependency, you can see in the example above that you still need to prefix the referencing of entities from the Week package by the name of the package. (If we had included a **use** Week clause, then such a prefix would not have been necessary.)

Accessing entities from a package uses the dot notation, A.B, which is the same notation as the one used to access record fields.

A **with** clause can *only* appear in the prelude of a compilation unit (i.e., before the reserved word, such as **procedure**, that marks the beginning of the unit). It is not allowed anywhere else. This rule is only needed for methodological reasons: the person reading your code should be able to see immediately which units the code depends on.

In other languages

Packages look similar to, but are semantically very different from, header files in C/C++.

- The first and most important distinction is that packages are a language-level mechanism. This is in contrast to a **#include**'d header file, which is a functionality of the C preprocessor.
- An immediate consequence is that the **with** construct is a semantic inclusion mechanism, not a text inclusion mechanism. Hence, when you **with** a package, you are saying to the compiler "I'm depending on this semantic unit", and not "include this bunch of text in place here".
- The effect of a package thus does not vary depending on where it has been **withed** from. Contrast this with C/C++, where the meaning of the included text depends on the context in which the **#include** appears.

This allows compilation/recompilation to be more efficient. It also allows tools like IDEs to have correct information about the semantics of a program. In turn, this allows better tooling in general, and code that is more analyzable, even by humans.

An important benefit of Ada **with** clauses when compared to **#include** is that it is stateless. The order of **with** and **use** clauses does not matter, and can be changed without side effects.

In the GNAT toolchain

The Ada language standard does not mandate any particular relationship between source files and packages; for example, in theory you can put all your code in one file, or use your own file naming conventions. In practice, however, an implementation will have specific rules. With GNAT, each top-level compilation unit needs to go into a separate file. In the example above, the Week package will be in an **.ads** file (for Ada specification), and the Main procedure will be in an **.adb** file (for Ada body).

4.2 Using a package

As we have seen above, the **with** clause indicates a dependency on another package. However, every reference to an entity coming from the `Week` package had to be prefixed by the full name of the package. It is possible to make every entity of a package visible directly in the current scope, using the **use** clause.

In fact, we have been using the **use** clause since almost the beginning of this tutorial.

Listing 36: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2      --      ^ Make every entity of the
3      --      Ada.Text_IO package
4      --      directly visible.
5  with Week;
6
7  procedure Main is
8      use Week;
9      -- Make every entity of the Week
10     -- package directly visible.
11  begin
12     Put_Line ("First day of the week is " & Mon);
13  end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Week
MD5: ea54077d4ae165b28ae8facfe8ba2db7

Runtime output

First day of the week is Monday

As you can see in the example above:

- `Put_Line` is a subprogram that comes from the `Ada.Text_IO` package. We can reference it directly because we have **used** the package at the top of the `Main` unit.
- Unlike **with** clauses, a **use** clause can be placed either in the prelude, or in any declarative region. In the latter case the **use** clause will have an effect in its containing lexical scope.

4.3 Package body

In the simple example above, the `Week` package only has declarations and no body. That's not a mistake: in a package specification, which is what is illustrated above, you cannot declare bodies. Those have to be in the package body.

Listing 37: operations.ads

```

1  package Operations is
2
3      -- Declaration
4      function Increment_By
5          (I      : Integer;
6           Incr   : Integer := 0) return Integer;
7
8      function Get_Increment_Value return Integer;
9
10 end Operations;
```


Listing 38: operations.adb

```

1 package body Operations is
2
3   Last_Increment : Integer := 1;
4
5   function Increment_By
6     (I      : Integer;
7      Incr   : Integer := 0) return Integer is
8   begin
9     if Incr /= 0 then
10      Last_Increment := Incr;
11    end if;
12
13    return I + Last_Increment;
14  end Increment_By;
15
16  function Get_Increment_Value return Integer is
17  begin
18    return Last_Increment;
19  end Get_Increment_Value;
20
21 end Operations;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Operations
MD5: 2adfb64e825605c74fecf6c9d45c8437

Here we can see that the body of the `Increment_By` function has to be declared in the body. Coincidentally, introducing a body allows us to put the `Last_Increment` variable in the body, and make them inaccessible to the user of the `Operations` package, providing a first form of encapsulation.

This works because entities declared in the body are *only* visible in the body.

This example shows how `Last_Increment` is used indirectly:

Listing 39: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Operations;
3
4 procedure Main is
5   use Operations;
6
7   I : Integer := 0;
8   R : Integer;
9
10  procedure Display_Update_Values is
11    Incr : constant Integer :=
12      Get_Increment_Value;
13  begin
14    Put_Line (Integer'Image (I)
15      & " incremented by "
16      & Integer'Image (Incr)
17      & " is "
18      & Integer'Image (R));
19    I := R;
20  end Display_Update_Values;
21 begin
22   R := Increment_By (I);

```

(continues on next page)

(continued from previous page)

```

23   Display_Update_Values;
24   R := Increment_By (I);
25   Display_Update_Values;
26
27   R := Increment_By (I, 5);
28   Display_Update_Values;
29   R := Increment_By (I);
30   Display_Update_Values;
31
32   R := Increment_By (I, 10);
33   Display_Update_Values;
34   R := Increment_By (I);
35   Display_Update_Values;
36 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Operations
MD5: 76190b1261a9652cfb7986ecec191e37

Runtime output

```

0 incremented by 1 is 1
1 incremented by 1 is 2
2 incremented by 5 is 7
7 incremented by 5 is 12
12 incremented by 10 is 22
22 incremented by 10 is 32
```

4.4 Child packages

Packages can be used to create hierarchies. We achieve this by using child packages, which extend the functionality of their parent package. One example of a child package that we've been using so far is the `Ada.Text_IO` package. Here, the parent package is called `Ada`, while the child package is called `Text_IO`. In the previous examples, we've been using the `Put_Line` procedure from the `Text_IO` child package.

Important

Ada also supports nested packages. However, since they can be more complicated to use, the recommendation is to use child packages instead. Nested packages will be covered in the advanced course.

Let's begin our discussion on child packages by taking our previous `Week` package:

Listing 40: `week.ads`

```

1 package Week is
2
3   Mon : constant String := "Monday";
4   Tue : constant String := "Tuesday";
5   Wed : constant String := "Wednesday";
6   Thu : constant String := "Thursday";
7   Fri : constant String := "Friday";
8   Sat : constant String := "Saturday";
9   Sun : constant String := "Sunday";
```

(continues on next page)

(continued from previous page)

```
10
11 end Week;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: 0fa033dc8fe2b9741483de273354e7ee

If we want to create a child package for Week, we may write:

Listing 41: week-child.ads

```
1 package Week.Child is
2
3     function Get_First_Of_Week return String;
4
5 end Week.Child;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: a7db38e772cf6153b5eb95069517e833

Here, Week is the parent package and Child is the child package. This is the corresponding package body of Week.Child:

Listing 42: week-child.adb

```
1 package body Week.Child is
2
3     function Get_First_Of_Week return String is
4     begin
5         return Mon;
6     end Get_First_Of_Week;
7
8 end Week.Child;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: 04dad82685ad9f0231c3084266b0af83

In the implementation of the Get_First_Of_Week function, we can use the Mon string directly, even though it was declared in the parent package Week. We don't write **with** Week here because all elements from the specification of the Week package — such as Mon, Tue and so on — are visible in the child package Week.Child.

Now that we've completed the implementation of the Week.Child package, we can use elements from this child package in a subprogram by simply writing **with** Week.Child. Similarly, if we want to use these elements directly, we write **use** Week.Child in addition. For example:

Listing 43: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week.Child;  use Week.Child;
3
4 procedure Main is
5 begin
6     Put_Line ("First day of the week is ")
```

(continues on next page)

(continued from previous page)

```

7      & Get_First_Of_Week);
8  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: e2f5c6ad3a92da4cb04ee7ec12293df4

Runtime output

First day of the week is Monday

4.4.1 Child of a child package

So far, we've seen a two-level package hierarchy. But the hierarchy that we can potentially create isn't limited to that. For instance, we could extend the hierarchy of the previous source code example by declaring a `Week.Child.Grandchild` package. In this case, `Week.Child` would be the parent of the `Grandchild` package. Let's consider this implementation:

Listing 44: week-child-grandchild.ads

```

1  package Week.Child.Grandchild is
2
3      function Get_Second_Of_Week return String;
4
5  end Week.Child.Grandchild;

```

Listing 45: week-child-grandchild.adb

```

1  package body Week.Child.Grandchild is
2
3      function Get_Second_Of_Week return String is
4      begin
5          return Tue;
6      end Get_Second_Of_Week;
7
8  end Week.Child.Grandchild;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: 03ee5932a68212b2e501370212508ab1

We can use this new `Grandchild` package in our test application in the same way as before: we can reuse the previous test application and adapt the **with** and **use**, and the function call. This is the updated code:

Listing 46: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Week.Child.Grandchild;
4  use Week.Child.Grandchild;
5
6  procedure Main is
7  begin
8      Put_Line ("Second day of the week is "
9              & Get_Second_Of_Week);
10 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: 29ee409c8131bd9529c6bf6e366bb390

Runtime output

Second day of the week is Tuesday

Again, this isn't the limit for the package hierarchy. We could continue to extend the hierarchy of the previous example by implementing a `Week.Child.Grandchild`. `Grand_grandchild` package.

4.4.2 Multiple children

So far, we've seen a single child package of a parent package. However, a parent package can also have multiple children. We could extend the example above and implement a `Week.Child_2` package. For example:

Listing 47: week-child_2.ads

```
1 package Week.Child_2 is
2
3     function Get_Last_Of_Week return String;
4
5 end Week.Child_2;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: bd3f63cacd142d9885600f4000b4573b

Here, `Week` is still the parent package of the `Child` package, but it's also the parent of the `Child_2` package. In the same way, `Child_2` is obviously one of the child packages of `Week`.

This is the corresponding package body of `Week.Child_2`:

Listing 48: week-child_2.adb

```
1 package body Week.Child_2 is
2
3     function Get_Last_Of_Week return String is
4     begin
5         return Sun;
6     end Get_Last_Of_Week;
7
8 end Week.Child_2;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: c2c03e4cb1daff02dd6076c2956ef2aa

We can now reference both children in our test application:

Listing 49: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week.Child; use Week.Child;
3 with Week.Child_2; use Week.Child_2;
4
5 procedure Main is
```

(continues on next page)

(continued from previous page)

```

6 begin
7   Put_Line ("First day of the week is "
8             & Get_First_Of_Week);
9   Put_Line ("Last day of the week is "
10            & Get_Last_Of_Week);
11 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Child_Packages
MD5: 6a91f239fb2a2d8c702409c22467a424

Runtime output

First day of the week is Monday
Last day of the week is Sunday

4.4.3 Visibility

In the previous section, we've seen that elements declared in a parent package specification are visible in the child package. This is, however, not the case for elements declared in the package body of a parent package.

Let's consider the package `Book` and its child `Additional_Operations`:

Listing 50: `book.ads`

```

1 package Book is
2
3   Title : constant String :=
4         "Visible for my children";
5
6   function Get_Title return String;
7
8   function Get_Author return String;
9
10 end Book;

```

Listing 51: `book-additional_operations.ads`

```

1 package Book.Additional_Operations is
2
3   function Get_Extended_Title return String;
4
5   function Get_Extended_Author return String;
6
7 end Book.Additional_Operations;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Visibility
MD5: a0d67cff9aeff288709391d16306df00

This is the body of both packages:

Listing 52: `book.adb`

```

1 package body Book is
2
3   Author : constant String :=

```

(continues on next page)

(continued from previous page)

```

4     "Author not visible for my children";
5
6     function Get_Title return String is
7     begin
8         return Title;
9     end Get_Title;
10
11    function Get_Author return String is
12    begin
13        return Author;
14    end Get_Author;
15
16 end Book;

```

Listing 53: book-additional_operations.adb

```

1 package body Book.Additional_Operations is
2
3     function Get_Extended_Title return String is
4     begin
5         return "Book Title: " & Title;
6     end Get_Extended_Title;
7
8     function Get_Extended_Author return String is
9     begin
10        -- "Author" string declared in the body
11        -- of the Book package is not visible
12        -- here. Therefore, we cannot write:
13        --
14        -- return "Book Author: " & Author;
15
16        return "Book Author: Unknown";
17    end Get_Extended_Author;
18
19 end Book.Additional_Operations;

```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Visibility
MD5: 68b7490da12bafae0aa6fe0ab76c6b1c

In the implementation of the `Get_Extended_Title`, we're using the `Title` constant from the parent package `Book`. However, as indicated in the comments of the `Get_Extended_Author` function, the `Author` string — which we declared in the body of the `Book` package — isn't visible in the `Book.Additional_Operations` package. Therefore, we cannot use it to implement the `Get_Extended_Author` function.

We can, however, use the `Get_Author` function from `Book` in the implementation of the `Get_Extended_Author` function to retrieve this string. Likewise, we can use this strategy to implement the `Get_Extended_Title` function. This is the adapted code:

Listing 54: book-additional_operations.adb

```

1 package body Book.Additional_Operations is
2
3     function Get_Extended_Title return String is
4     begin
5         return "Book Title: " & Get_Title;
6     end Get_Extended_Title;
7

```

(continues on next page)

(continued from previous page)

```

8   function Get_Extended_Author return String is
9   begin
10    return "Book Author: " & Get_Author;
11  end Get_Extended_Author;
12
13 end Book.Additional_Operations;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Visibility
MD5: b00c187cb54d3fcb9574726028c1efc6

This is a simple test application for the packages above:

Listing 55: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Book.Additional_Operations;
4  use Book.Additional_Operations;
5
6  procedure Main is
7  begin
8    Put_Line (Get_Extended_Title);
9    Put_Line (Get_Extended_Author);
10 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Visibility
MD5: bdc75987fe61e9401b400f8704890ebe

Runtime output

Book Title: Visible for my children
Book Author: Author not visible for my children

By declaring elements in the body of a package, we can implement encapsulation in Ada. Those elements will only be visible in the package body, but nowhere else. This isn't, however, the only way to achieve encapsulation in Ada: we'll discuss other approaches in the *Privacy* (page 115) chapter.

4.5 Renaming

Previously, we've mentioned that *subprograms can be renamed* (page 31). We can rename packages, too. Again, we use the **renames** keyword for that. The following example renames the `Ada.Text_IO` package as `TI0`:

Listing 56: main.adb

```

1  with Ada.Text_IO;
2
3  procedure Main is
4    package TI0 renames Ada.Text_IO;
5  begin
6    TI0.Put_Line ("Hello");
7  end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Rename_Text_IO
MD5: 33652dd004ef33d95c168ab8893cd412

Runtime output

Hello

We can use renaming to improve the readability of our code by using shorter package names. In the example above, we write `TI0.Put_Line` instead of the longer version (`Ada.Text_IO.Put_Line`). This approach is especially useful when we don't **use** packages and want to avoid that the code becomes too verbose.

Note we can also rename subprograms and objects inside packages. For instance, we could have just renamed the `Put_Line` procedure in the source code example above:

Listing 57: main.adb

```
1 with Ada.Text_IO;  
2  
3 procedure Main is  
4     procedure Say (Something : String)  
5         renames Ada.Text_IO.Put_Line;  
6 begin  
7     Say ("Hello");  
8 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Modular_Programming.Rename_Put_Line
MD5: f30174ff29eb01f33bc95f1787f9f1dc

Runtime output

Hello

In this example, we rename the `Put_Line` procedure to `Say`.

STRONGLY TYPED LANGUAGE

Ada is a strongly typed language. It is interestingly modern in that respect: strong static typing has been increasing in popularity in programming language design, owing to factors such as the growth of statically typed functional programming, a big push from the research community in the typing domain, and many practical languages with strong type systems.

5.1 What is a type?

In statically typed languages, a type is mainly (but not only) a *compile time* construct. It is a construct to enforce invariants about the behavior of a program. Invariants are unchangeable properties that hold for all variables of a given type. Enforcing them ensures, for example, that variables of a data type never have invalid values.

A type is used to reason about the *objects* a program manipulates (an object is a variable or a constant). The aim is to classify objects by what you can accomplish with them (i.e., the operations that are permitted), and this way you can reason about the correctness of the objects' values.

5.2 Integers

A nice feature of Ada is that you can define your own integer types, based on the requirements of your program (i.e., the range of values that makes sense). In fact, the definitional mechanism that Ada provides forms the semantic basis for the predefined integer types. There is no "magical" built-in type in that regard, which is unlike most languages, and arguably very elegant.

Listing 58: integer_type_example.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Integer_Type_Example is
4   -- Declare a signed integer type,
5   -- and give the bounds
6   type My_Int is range -1 .. 20;
7   --           ^ High bound
8   --           ^ Low bound
9
10  -- Like variables, type declarations can
11  -- only appear in declarative regions.
12 begin
13   for I in My_Int loop
14     Put_Line (My_Int'Image (I));
15     --           ^ 'Image attribute
16     --           converts a value
17     --           to a String.
```

(continues on next page)

(continued from previous page)

```

18   end loop;
19 end Integer_Type_Example;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Integer_Type_Example
 MD5: 1d82fa54b604944fdd8652cbf84f4ff2

Runtime output

```

-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

```

This example illustrates the declaration of a signed integer type, and several things we can do with them.

Every type declaration in Ada starts with the **type** keyword (except for *task types* (page 162)). After the type, we can see a range that looks a lot like the ranges that we use in **for** loops, that defines the low and high bound of the type. Every integer in the inclusive range of the bounds is a valid value for the type.

Ada integer types

In Ada, an integer type is not specified in terms of its machine representation, but rather by its range. The compiler will then choose the most appropriate representation.

Another point to note in the above example is the `My_Int'Image` (I) expression. The `Name'Attribute` (optional params) notation is used for what is called an attribute in Ada. An attribute is a built-in operation on a type, a value, or some other program entity. It is accessed by using a ' symbol (the ASCII apostrophe).

Ada has several types available as "built-ins"; **Integer** is one of them. Here is how **Integer** might be defined for a typical processor:

```

type Integer is
  range -(2 ** 31) .. +(2 ** 31 - 1);

```

****** is the exponent operator, which means that the first valid value for **Integer** is -2^{31} , and the last valid value is $2^{31} - 1$.

Ada does not mandate the range of the built-in type **Integer**. An implementation for a 16-bit target would likely choose the range -2^{15} through $2^{15} - 1$.

5.2.1 Operational semantics

Unlike some other languages, Ada requires that operations on integers should be checked for overflow.

Listing 59: main.adb

```

1 procedure Main is
2   A : Integer := Integer'Last;
3   B : Integer;
4 begin
5   B := A + 5;
6   -- This operation will overflow, eg. it
7   -- will raise an exception at run time.
8 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Overflow_Check
MD5: bddd15b394f043442024899d12b982fb

Build output

```

main.adb:5:11: warning: value not in range of type "Standard.Integer" [enabled by
↳default]
main.adb:5:11: warning: Constraint_Error will be raised at run time [enabled by
↳default]
```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:5 overflow check failed
```

There are two types of overflow checks:

- Machine-level overflow, when the result of an operation exceeds the maximum value (or is less than the minimum value) that can be represented in the storage reserved for an object of the type, and
- Type-level overflow, when the result of an operation is outside the range defined for the type.

Mainly for efficiency reasons, while machine-level overflow always results in an exception, type-level overflows will only be checked at specific boundaries, like assignment:

Listing 60: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type My_Int is range 1 .. 20;
5   A : My_Int := 12;
6   B : My_Int := 15;
7   M : My_Int := (A + B) / 2;
8   -- No overflow here, overflow checks
9   -- are done at specific boundaries.
10 begin
11   for I in 1 .. M loop
12     Put_Line ("Hello, World!");
```

(continues on next page)

(continued from previous page)

```

13   end loop;
14   -- Loop body executed 13 times
15 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Overflow_Check_2
MD5: d24283cbb42c0be5b5fa215eb16ad2e7

Runtime output

```

Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!

```

Type-level overflow will only be checked at specific points in the execution. The result, as we see above, is that you might have an operation that overflows in an intermediate computation, but no exception will be raised because the final result does not overflow.

5.3 Unsigned types

Ada also features unsigned Integer types. They're called *modular* types in Ada parlance. The reason for this designation is due to their behavior in case of overflow: They simply "wrap around", as if a modulo operation was applied.

For machine sized modular types, for example a modulus of 2^{32} , this mimics the most common implementation behavior of unsigned types. However, an advantage of Ada is that the modulus is more general:

Listing 61: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Mod_Int is mod 2 ** 5;
5          --           ^ Range is 0 .. 31
6
7      A : constant Mod_Int := 20;
8      B : constant Mod_Int := 15;
9
10     M : constant Mod_Int := A + B;
11     -- No overflow here,
12     -- M = (20 + 15) mod 32 = 3
13 begin
14     for I in 1 .. M loop
15         Put_Line ("Hello, World!");
16     end loop;
17 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Unsigned_Types
 MD5: df4efee4eb29e7ea15a0cf961b600dd5

Runtime output

```
Hello, World!
Hello, World!
Hello, World!
```

Unlike in C/C++, since this wraparound behavior is guaranteed by the Ada specification, you can rely on it to implement portable code. Also, being able to leverage the wrapping on arbitrary bounds is very useful — the modulus does not need to be a power of 2 — to implement certain algorithms and data structures, such as [ring buffers](#)¹⁰.

5.4 Enumerations

Enumeration types are another nicety of Ada's type system. Unlike C's enums, they are *not* integers, and each new enumeration type is incompatible with other enumeration types. Enumeration types are part of the bigger family of discrete types, which makes them usable in certain situations that we will describe later but one context that we have already seen is a case statement.

Listing 62: enumeration_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Enumeration_Example is
4      type Days is (Monday, Tuesday, Wednesday,
5                    Thursday, Friday,
6                    Saturday, Sunday);
7      -- An enumeration type
8  begin
9      for I in Days loop
10         case I is
11             when Saturday .. Sunday =>
12                 Put_Line ("Week end!");
13
14             when Monday .. Friday =>
15                 Put_Line ("Hello on "
16                           & Days'Image (I));
17                 -- 'Image attribute, works on
18                 -- enums too
19         end case;
20     end loop;
21 end Enumeration_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Enumeration_Example
 MD5: 45d6c83992af4fb6d5015d5f22cb7113

Runtime output

```
Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
```

(continues on next page)

¹⁰ https://en.wikipedia.org/wiki/Circular_buffer

(continued from previous page)

```
Week end!  
Week end!
```

Enumeration types are powerful enough that, unlike in most languages, they're used to define the standard Boolean type:

```
type Boolean is (False, True);
```

As mentioned previously, every "built-in" type in Ada is defined with facilities generally available to the user.

5.5 Floating-point types

5.5.1 Basic properties

Like most languages, Ada supports floating-point types. The most commonly used floating-point type is **Float**:

Listing 63: floating_point_demo.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Floating_Point_Demo is  
4   A : constant Float := 2.5;  
5 begin  
6   Put_Line ("The value of A is "  
7             & Float'Image (A));  
8 end Floating_Point_Demo;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Demo
MD5: 06998775497b68b742700138faecbb6a

Runtime output

```
The value of A is 2.50000E+00
```

The application will display 2.5 as the value of A.

The Ada language does not specify the precision (number of decimal digits in the mantissa) for Float; on a typical 32-bit machine the precision will be 6.

All common operations that could be expected for floating-point types are available, including absolute value and exponentiation. For example:

Listing 64: floating_point_operations.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Floating_Point_Operations is  
4   A : Float := 2.5;  
5 begin  
6   A := abs (A - 4.5);  
7   Put_Line ("The value of A is "  
8             & Float'Image (A));  
9  
10  A := A ** 2 + 1.0;  
11  Put_Line ("The value of A is "
```

(continues on next page)

(continued from previous page)

```

12         & Float'Image (A));
13 end Floating_Point_Operations;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Operations
MD5: c280e0f23e020aaee1a8777e7fb4c242

Runtime output

```

The value of A is  2.00000E+00
The value of A is  5.00000E+00

```

The value of A is 2.0 after the first operation and 5.0 after the second operation.

In addition to **Float**, an Ada implementation may offer data types with higher precision such as **Long_Float** and **Long_Long_Float**. Like **Float**, the standard does not indicate the exact precision of these types: it only guarantees that the type **Long_Float**, for example, has at least the precision of **Float**. In order to guarantee that a certain precision requirement is met, we can define custom floating-point types, as we will see in the next section.

5.5.2 Precision of floating-point types

Ada allows the user to specify the precision for a floating-point type, expressed in terms of decimal digits. Operations on these custom types will then have at least the specified precision. The syntax for a simple floating-point type declaration is:

```
type T is digits <number_of_decimal_digits>;
```

The compiler will choose a floating-point representation that supports the required precision. For example:

Listing 65: custom_floating_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Custom_Floating_Types is
4      type T3 is digits 3;
5      type T15 is digits 15;
6      type T18 is digits 18;
7  begin
8      Put_Line ("T3 requires "
9                & Integer'Image (T3'Size)
10               & " bits");
11      Put_Line ("T15 requires "
12                & Integer'Image (T15'Size)
13               & " bits");
14      Put_Line ("T18 requires "
15                & Integer'Image (T18'Size)
16               & " bits");
17  end Custom_Floating_Types;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Custom_Floating_Types
MD5: 3c23738f13e081038996c533da8fb723

Runtime output


```
T3 requires 32 bits
T15 requires 64 bits
T18 requires 128 bits
```

In this example, the attribute `'Size` is used to retrieve the number of bits used for the specified data type. As we can see by running this example, the compiler allocates 32 bits for T3, 64 bits for T15 and 128 bits for T18. This includes both the mantissa and the exponent.

The number of digits specified in the data type is also used in the format when displaying floating-point variables. For example:

Listing 66: `display_custom_floating_types.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Custom_Floating_Types is
4   type T3 is digits 3;
5   type T18 is digits 18;
6
7   C1 : constant := 1.0e-4;
8
9   A : constant T3 := 1.0 + C1;
10  B : constant T18 := 1.0 + C1;
11 begin
12   Put_Line ("The value of A is "
13             & T3'Image (A));
14   Put_Line ("The value of B is "
15             & T18'Image (B));
16 end Display_Custom_Floating_Types;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Display_Custom_Floating_Types
MD5: 58ec2660388a7f05e139f73e94303cf1
```

Runtime output

```
The value of A is 1.00E+00
The value of B is 1.000100000000000000E+00
```

As expected, the application will display the variables according to specified precision (1.00E+00 and 1.000100000000000000E+00).

5.5.3 Range of floating-point types

In addition to the precision, a range can also be specified for a floating-point type. The syntax is similar to the one used for integer data types — using the `range` keyword. This simple example creates a new floating-point type based on the type `Float`, for a normalized range between `-1.0` and `1.0`:

Listing 67: `floating_point_range.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Range is
4   type T_Norm is new Float range -1.0 .. 1.0;
5   A : T_Norm;
6 begin
7   A := 1.0;
8   Put_Line ("The value of A is "
```

(continues on next page)

(continued from previous page)

```

9         & T_Norm'Image (A));
10 end Floating_Point_Range;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Range
MD5: b43d596682aa0fa11124a3a3d0596abc

Runtime output

The value of A is 1.00000E+00

The application is responsible for ensuring that variables of this type stay within this range; otherwise an exception is raised. In this example, the exception `Constraint_Error` is raised when assigning `2.0` to the variable `A`:

Listing 68: floating_point_range_exception.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Range_Exception is
4     type T_Norm is new Float range -1.0 .. 1.0;
5     A : T_Norm;
6 begin
7     A := 2.0;
8     Put_Line ("The value of A is "
9         & T_Norm'Image (A));
10 end Floating_Point_Range_Exception;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Floating_Point_Range_
↳Exception
MD5: ecda66589ba28e453956dca159ea5f0d

Build output

```

floating_point_range_exception.adb:7:09: warning: value not in range of type "T_
↳Norm" defined at line 4 [enabled by default]
floating_point_range_exception.adb:7:09: warning: Constraint_Error will be raised_
↳at run time [enabled by default]

```

Runtime output

raised CONSTRAINT_ERROR : floating_point_range_exception.adb:7 range check failed

Ranges can also be specified for custom floating-point types. For example:

Listing 69: custom_range_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics; use Ada.Numerics;
3
4 procedure Custom_Range_Types is
5     type T6_Inv_Trig is
6         digits 6 range -Pi / 2.0 .. Pi / 2.0;
7 begin
8     null;
9 end Custom_Range_Types;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Custom_Range_Types
MD5: 7b62abc869290a30e351163f670059e0

In this example, we are defining a type called `T6_Inv_Trig`, which has a range from $-\pi / 2$ to $\pi / 2$ with a minimum precision of 6 digits. (`Pi` is defined in the predefined package `Ada.Numerics`.)

5.6 Strong typing

As noted earlier, Ada is strongly typed. As a result, different types of the same family are incompatible with each other; a value of one type cannot be assigned to a variable from the other type. For example:

Listing 70: `illegal_example.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Illegal_Example is
4   -- Declare two different floating point types
5   type Meters is new Float;
6   type Miles is new Float;
7
8   Dist_Imperial : Miles;
9
10  -- Declare a constant
11  Dist_Metric : constant Meters := 1000.0;
12 begin
13   -- Not correct: types mismatch
14   Dist_Imperial := Dist_Metric * 621.371e-6;
15   Put_Line (Miles'Image (Dist_Imperial));
16 end Illegal_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Imperial_Metric_Error
MD5: e28e341c5eda9b3b4cef691fa24b7f7e

Build output

```
illegal_example.adb:14:33: error: expected type "Miles" defined at line 6
illegal_example.adb:14:33: error: found type "Meters" defined at line 5
gprbuild: *** compilation phase failed
```

A consequence of these rules is that, in the general case, a "mixed mode" expression like `2 * 3.0` will trigger a compilation error. In a language like C or Python, such expressions are made valid by implicit conversions. In Ada, such conversions must be made explicit:

Listing 71: `conv.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure Conv is
3   type Meters is new Float;
4   type Miles is new Float;
5   Dist_Imperial : Miles;
6   Dist_Metric : constant Meters := 1000.0;
7 begin
8   Dist_Imperial :=
9     Miles (Dist_Metric) * 621.371e-6;
10  -- ~~~~~
```

(continues on next page)

(continued from previous page)

```

11  --    Type conversion, from Meters to Miles
12  --    Now the code is correct
13
14  Put_Line (Miles'Image (Dist_Imperial));
15 end Conv;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Imperial_Metric
MD5: e455641e86227e80e5f920b5af6315d4

Runtime output

6.21371E-01

Of course, we probably do not want to write the conversion code every time we convert from meters to miles. The idiomatic Ada way in that case would be to introduce conversion functions along with the types.

Listing 72: conv.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Conv is
4      type Meters is new Float;
5      type Miles  is new Float;
6
7      -- Function declaration, like procedure
8      -- but returns a value.
9      function To_Miles (M : Meters) return Miles is
10         -- ^ Return type
11     begin
12         return Miles (M) * 621.371e-6;
13     end To_Miles;
14
15     Dist_Imperial : Miles;
16     Dist_Metric   : constant Meters := 1000.0;
17 begin
18     Dist_Imperial := To_Miles (Dist_Metric);
19     Put_Line (Miles'Image (Dist_Imperial));
20 end Conv;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Imperial_Metric_Func
MD5: 661737fa9f130ac3070210bbf6f08214

Runtime output

6.21371E-01

If you write a lot of numeric code, having to explicitly provide such conversions might seem painful at first. However, this approach brings some advantages. Notably, you can rely on the absence of implicit conversions, which will in turn prevent some subtle errors.

** In other languages**

In C, for example, the rules for implicit conversions may not always be completely obvious. In Ada, however, the code will always do exactly what it seems to do. For example:

```
int a = 3, b = 2;
float f = a / b;
```

This code will compile fine, but the result of `f` will be 1.0 instead of 1.5, because the compiler will generate an integer division (three divided by two) that results in one. The software developer must be aware of data conversion issues and use an appropriate casting:

```
int a = 3, b = 2;
float f = (float)a / b;
```

In the corrected example, the compiler will convert both variables to their corresponding floating-point representation before performing the division. This will produce the expected result.

This example is very simple, and experienced C developers will probably notice and correct it before it creates bigger problems. However, in more complex applications where the type declaration is not always visible — e.g. when referring to elements of a **struct** — this situation might not always be evident and quickly lead to software defects that can be harder to find.

The Ada compiler, in contrast, will always reject code that mixes floating-point and integer variables without explicit conversion. The following Ada code, based on the erroneous example in C, will not compile:

Listing 73: main.adb

```
1 procedure Main is
2   A : Integer := 3;
3   B : Integer := 2;
4   F : Float;
5 begin
6   F := A / B;
7 end Main;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Implicit_Cast
MD5: 38a8fcc6608c22e22940052ab8dd62f4
```

Build output

```
main.adb:6:11: error: expected type "Standard.Float"
main.adb:6:11: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed
```

The offending line must be changed to `F := Float (A) / Float (B);` in order to be accepted by the compiler.

You can use Ada's strong typing to help enforce invariants in your code, as in the example above: Since Miles and Meters are two different types, you cannot mistakenly convert an instance of one to an instance of the other.

5.7 Derived types

In Ada you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing.

Listing 74: main.adb

```

1  procedure Main is
2      -- ID card number type,
3      -- incompatible with Integer.
4      type Social_Security_Number is new Integer
5          range 0 .. 999_99_9999;
6      -- ^ Since a SSN has 9 digits
7      --     max., and cannot be
8      --     negative, we enforce
9      --     a validity constraint.
10
11     SSN : Social_Security_Number :=
12         555_55_5555;
13     -- ^ You can put underscores as
14     --     formatting in any number.
15
16     I    : Integer;
17
18     -- The value -1 below will cause a
19     -- runtime error and a compile time
20     -- warning with GNAT.
21     Invalid : Social_Security_Number := -1;
22 begin
23     -- Illegal, they have different types:
24     I := SSN;
25
26     -- Likewise illegal:
27     SSN := I;
28
29     -- OK with explicit conversion:
30     I := Integer (SSN);
31
32     -- Likewise OK:
33     SSN := Social_Security_Number (I);
34 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Derived_Types
MD5: 63445601ddb5e52dceab095d3305623a

Build output

```

main.adb:21:40: warning: value not in range of type "Social_Security_Number"
↳ defined at line 4 [enabled by default]
main.adb:21:40: warning: Constraint_Error will be raised at run time [enabled by
↳ default]
main.adb:24:09: error: expected type "Standard.Integer"
main.adb:24:09: error: found type "Social_Security_Number" defined at line 4
main.adb:27:11: error: expected type "Social_Security_Number" defined at line 4
main.adb:27:11: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed

```

The type `Social_Security` is said to be a *derived type*; its *parent type* is `Integer`.

As illustrated in this example, you can refine the valid range when defining a derived scalar type (such as integer, floating-point and enumeration).

The syntax for enumerations uses the `range` `<range>` syntax:

Listing 75: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   type Days is (Monday, Tuesday, Wednesday,
5                 Thursday, Friday,
6                 Saturday, Sunday);
7
8   type Weekend_Days is new
9     Days range Saturday .. Sunday;
10  -- New type, where only Saturday and Sunday
11  -- are valid literals.
12 begin
13   null;
14 end Greet;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Days
MD5: 853b5c1576961c7c20d4306275122364
```

5.8 Subtypes

As we are starting to see, types may be used in Ada to enforce constraints on the valid range of values. However, we sometimes want to enforce constraints on some values while staying within a single type. This is where subtypes come into play. A subtype does not introduce a new type.

Listing 76: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   type Days is (Monday, Tuesday, Wednesday,
5                 Thursday, Friday,
6                 Saturday, Sunday);
7
8   -- Declaration of a subtype
9   subtype Weekend_Days is
10     Days range Saturday .. Sunday;
11   -- ^ Constraint of the subtype
12
13   M : Days := Sunday;
14
15   S : Weekend_Days := M;
16   -- No error here, Days and Weekend_Days
17   -- are of the same type.
18 begin
19   for I in Days loop
20     case I is
21       -- Just like a type, a subtype can
22       -- be used as a range
23       when Weekend_Days =>
24         Put_Line ("Week end!");
25       when others =>
26         Put_Line ("Hello on "
27                   & Days'Image (I));
28     end case;
```

(continues on next page)

(continued from previous page)

```

29   end loop;
30 end Greet;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Days_Subtype
MD5: 8ee7127d152a8b2c9d0ac74d05fc2fc2

Runtime output

```

Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!

```

Several subtypes are predefined in the standard package in Ada, and are automatically available to you:

```

subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;

```

While subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised.

Listing 77: greet.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4      type Days is (Monday, Tuesday, Wednesday,
5                    Thursday, Friday,
6                    Saturday, Sunday);
7
8      subtype Weekend_Days is
9          Days range Saturday .. Sunday;
10
11     Day      : Days := Saturday;
12     Weekend  : Weekend_Days;
13 begin
14     Weekend := Day;
15     --      ^ Correct: Same type, subtype
16     --      constraints are respected
17     Weekend := Monday;
18     --      ^ Wrong value for the subtype
19     --      Compiles, but exception at runtime
20 end Greet;

```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Days_Subtype_Error
MD5: 84d42d276d26544f35edab5870459378

Build output

```

greet.adb:17:15: warning: value not in range of type "Weekend_Days" defined at
↳line 8 [enabled by default]
greet.adb:17:15: warning: Constraint_Error will be raised at run time [enabled by
↳default]

```


Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:17 range check failed
```

5.8.1 Subtypes as type aliases

Previously, we've seen that we can create new types by declaring e.g. **type Miles is new Float**. We could also create type aliases, which generate alternative names — *aliases* — for known types. Note that type aliases are sometimes called *type synonyms*.

We achieve this in Ada by using subtypes without new constraints. In this case, however, we don't get all of the benefits of Ada's strong type checking. Let's rewrite an example using type aliases:

Listing 78: undetected_imperial_metric_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Undetected_Imperial_Metric_Error is
4   -- Declare two type aliases
5   subtype Meters is Float;
6   subtype Miles is Float;
7
8   Dist_Imperial : Miles;
9
10  -- Declare a constant
11  Dist_Metric : constant Meters := 100.0;
12 begin
13   -- No conversion to Miles type required:
14   Dist_Imperial := (Dist_Metric * 1609.0)
15                   / 1000.0;
16
17   -- Not correct, but undetected:
18   Dist_Imperial := Dist_Metric;
19
20   Put_Line (Miles'Image (Dist_Imperial));
21 end Undetected_Imperial_Metric_Error;
```

Code block metadata

Project: Courses.Intro_To_Ada.Strongly_Typed_Language.Undetected_Imperial_Metric_Error
MD5: cdb8f949c69f3c480502b859dac298ee

Runtime output

```
1.00000E+02
```

In the example above, the fact that both Meters and Miles are subtypes of **Float** allows us to mix variables of both types without type conversion. This, however, can lead to all sorts of programming mistakes that we'd like to avoid, as we can see in the undetected error highlighted in the code above. In that example, the error in the assignment of a value in meters to a variable meant to store values in miles remains undetected because both Meters and Miles are subtypes of **Float**. Therefore, the recommendation is to use strong typing — via **type X is new Y** — for cases such as the one above.

There are, however, many situations where type aliases are useful. For example, in an application that uses floating-point types in multiple contexts, we could use type aliases to indicate additional meaning to the types or to avoid long variable names. For example, instead of writing:

```
Paid_Amount, Due_Amount : Float;
```

We could write:

```
subtype Amount is Float;

Paid, Due : Amount;
```

In other languages

In C, for example, we can use a **typedef** declaration to create a type alias. For example:

```
typedef float meters;
```

This corresponds to the declaration that we've seen above using subtypes. Other programming languages include this concept in similar ways. For example:

- C++: using meters = float;
- Swift: typealias Meters = Double
- Kotlin: typealias Meters = Double
- Haskell: type Meters = Float

Note, however, that subtypes in Ada correspond to type aliases if, and only if, they don't have new constraints. Thus, if we add a new constraint to a subtype declaration, we don't have a type alias anymore. For example, the following declaration *can't* be considered a type alias of **Float**:

```
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

Let's look at another example:

```
subtype Degree_Celsius is Float;

subtype Liquid_Water_Temperature is
  Degree_Celsius range 0.0 .. 100.0;

subtype Running_Water_Temperature is
  Liquid_Water_Temperature;
```

In this example, Liquid_Water_Temperature isn't an alias of Degree_Celsius, since it adds a new constraint that wasn't part of the declaration of the Degree_Celsius. However, we do have two type aliases here:

- Degree_Celsius is an alias of **Float**;
- Running_Water_Temperature is an alias of Liquid_Water_Temperature, even if Liquid_Water_Temperature itself has a constrained range.

RECORDS

So far, all the types we have encountered have values that are not decomposable: each instance represents a single piece of data. Now we are going to see our first class of composite types: records.

Records allow composing a value out of instances of other types. Each of those instances will be given a name. The pair consisting of a name and an instance of a specific type is called a field, or a component.

6.1 Record type declaration

Here is an example of a simple record declaration:

```
type Date is record
  -- The following declarations are
  -- components of the record
  Day   : Integer range 1 .. 31;
  Month : Months;
  -- You can add custom constraints
  -- on fields
  Year  : Integer range 1 .. 3000;
end record;
```

Fields look a lot like variable declarations, except that they are inside of a record definition. And as with variable declarations, you can specify additional constraints when supplying the subtype of the field.

```
type Date is record
  Day   : Integer range 1 .. 31;
  Month : Months := January;
  -- This component has a default value
  Year  : Integer range 1 .. 3000 := 2012;
  --
  --                                     Default value
end record;
```

Record components can have default values. When a variable having the record type is declared, a field with a default initialization will be automatically set to this value. The value can be any expression of the component type, and may be run-time computable.

In the remaining sections of this chapter, we see how to use record types. In addition to that, we discuss more about records in [another chapter](#) (page 101).

6.2 Aggregates

```
-- Positional components
Ada_Birthday    : Date := (10, December, 1815);

-- Named components
Leap_Day_2020   : Date := (Day    => 29,
                           Month => February,
                           Year  => 2020);
--                               ^ By name
```

Records have a convenient notation for expressing values, illustrated above. This notation is called aggregate notation, and the literals are called aggregates. They can be used in a variety of contexts that we will see throughout the course, one of which is to initialize records.

An aggregate is a list of values separated by commas and enclosed in parentheses. It is allowed in any context where a value of the record is expected.

Values for the components can be specified positionally, as in `Ada_Birthday` example, or by name, as in `Leap_Day_2020`. A mixture of positional and named values is permitted, but you cannot use a positional notation after a named one.

6.3 Component selection

To access components of a record instance, you use an operation that is called component selection. This is achieved by using the dot notation. For example, if we declare a variable `Some_Day` of the `Date` record type mentioned above, we can access the `Year` component by writing `Some_Day.Year`.

Let's look at an example:

Listing 79: `record_selection.adb`

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Record_Selection is
4
5      type Months is
6          (January, February, March, April,
7           May, June, July, August, September,
8           October, November, December);
9
10     type Date is record
11         Day    : Integer range 1 .. 31;
12         Month  : Months;
13         Year   : Integer range 1 .. 3000 := 2032;
14     end record;
15
16     procedure Display_Date (D : Date) is
17     begin
18         Put_Line ("Day:" & Integer'Image (D.Day)
19                  & ", Month: "
20                  & Months'Image (D.Month)
21                  & ", Year:"
22                  & Integer'Image (D.Year));
23     end Display_Date;
24
25     Some_Day : Date := (1, January, 2000);
26
```

(continues on next page)

(continued from previous page)

```

27 begin
28   Display_Date (Some_Day);
29
30   Put_Line ("Changing year...");
31   Some_Day.Year := 2001;
32
33   Display_Date (Some_Day);
34 end Record_Selection;

```

Code block metadata

Project: Courses.Intro_To_Ada.Records.Record_Selection
MD5: 79602cf4d011ba7423d07772b13e2b5a

Runtime output

```

Day: 1, Month: JANUARY, Year: 2000
Changing year...
Day: 1, Month: JANUARY, Year: 2001

```

As you can see in this example, we can use the dot notation in the expression `D.Year` or `Some_Day.Year` to access the information stored in that component, as well as to modify this information in assignments. To be more specific, when we use `D.Year` in the call to `Put_Line`, we're retrieving the information stored in that component. When we write `Some_Day.Year := 2001`, we're overwriting the information that was previously stored in the `Year` component of `Some_Day`.

6.4 Renaming

In previous chapters, we've discussed *subprogram* (page 31) and *package* (page 45) renaming. We can rename record components as well. Instead of writing the full component selection using the dot notation, we can declare an alias that allows us to access the same component. This is useful to simplify the implementation of a subprogram, for example.

We can rename record components by using the **renames** keyword in a variable declaration. For example:

```

Some_Day : Date;
Y        : Integer renames Some_Day.Year;

```

Here, `Y` is an alias, so that every time we using `Y`, we are really using the `Year` component of `Some_Day`.

Let's look at a complete example:

Listing 80: dates.ads

```

1 package Dates is
2
3   type Months is
4     (January, February, March, April,
5      May, June, July, August, September,
6      October, November, December);
7
8   type Date is record
9     Day   : Integer range 1 .. 31;
10    Month : Months;
11    Year  : Integer range 1 .. 3000 := 2032;
12 end record;

```

(continues on next page)

(continued from previous page)

```
13
14 procedure Increase_Month
15   (Some_Day : in out Date);
16
17 procedure Display_Month
18   (Some_Day : Date);
19
20 end Dates;
```

Listing 81: dates.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Dates is
4
5   procedure Increase_Month
6     (Some_Day : in out Date)
7   is
8     -- Renaming components from
9     -- the Date record
10    M : Months renames Some_Day.Month;
11    Y : Integer renames Some_Day.Year;
12
13    -- Renaming function (for Months
14    -- enumeration)
15    function Next (M : Months)
16      return Months
17      renames Months'Succ;
18  begin
19    if M = December then
20      M := January;
21      Y := Y + 1;
22    else
23      M := Next (M);
24    end if;
25  end Increase_Month;
26
27  procedure Display_Month
28    (Some_Day : Date)
29  is
30    -- Renaming components from
31    -- the Date record
32    M : Months renames Some_Day.Month;
33    Y : Integer renames Some_Day.Year;
34  begin
35    Put_Line ("Month: "
36              & Months'Image (M)
37              & ", Year:"
38              & Integer'Image (Y));
39  end Display_Month;
40
41 end Dates;
```

Listing 82: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Dates;       use Dates;
3
4 procedure Main is
5   D : Date := (1, January, 2000);
```

(continues on next page)

(continued from previous page)

```
6 begin
7   Display_Month (D);
8
9   Put_Line ("Increasing month...");
10  Increase_Month (D);
11
12  Display_Month (D);
13 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Record_Component_Renaming
MD5: 905390bd02b8417039052218800975a3

Runtime output

Month: JANUARY, Year: 2000
Increasing month...
Month: FEBRUARY, Year: 2000

We apply renaming to two components of the Date record in the implementation of the Increase_Month procedure. Then, instead of directly using Some_Day.Month and Some_Day.Year in the next operations, we simply use the renamed versions M and Y.

Note that, in the example above, we also rename Months 'Succ — which is the function that gives us the next month — to Next.

ARRAYS

Arrays provide another fundamental family of composite types in Ada.

7.1 Array type declaration

Arrays in Ada are used to define contiguous collections of elements that can be selected by indexing. Here's a simple example:

Listing 83: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   type My_Int is range 0 .. 1000;
5   type Index is range 1 .. 5;
6
7   type My_Int_Array is
8     array (Index) of My_Int;
9     --           ^ Type of elements
10    --           ^ Bounds of the array
11    Arr : My_Int_Array := (2, 3, 5, 7, 11);
12    --           ^ Array literal
13    --           (aggregate)
14
15    V : My_Int;
16 begin
17   for I in Index loop
18     V := Arr (I);
19     --           ^ Take the Ith element
20     Put (My_Int'Image (V));
21   end loop;
22   New_Line;
23 end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Greet
MD5: ffdd2ba2322b0946dfcac3a55bce5270

Runtime output

```
2 3 5 7 11
```

The first point to note is that we specify the index type for the array, rather than its size. Here we declared an integer type named `Index` ranging from `1` to `5`, so each array instance will have 5 elements, with the initial element at index 1 and the last element at index 5.

Although this example used an integer type for the index, Ada is more general: any discrete type is permitted to index an array, including *Enum types* (page 51). We will soon see what that means.

Another point to note is that querying an element of the array at a given index uses the same syntax as for function calls: that is, the array object followed by the index in parentheses.

Thus when you see an expression such as `A (B)`, whether it is a function call or an array subscript depends on what `A` refers to.

Finally, notice how we initialize the array with the `(2, 3, 5, 7, 11)` expression. This is another kind of aggregate in Ada, and is in a sense a literal expression for an array, in the same way that `3` is a literal expression for an integer. The notation is very powerful, with a number of properties that we will introduce later. A detailed overview appears in the notation of *aggregate types* (page 89).

Unrelated to arrays, the example also illustrated two procedures from `Ada.Text_IO`:

- `Put`, which displays a string without a terminating end of line
- `New_Line`, which outputs an end of line

Let's now delve into what it means to be able to use any discrete type to index into the array.

In other languages

Semantically, an array object in Ada is the entire data structure, and not simply a handle or pointer. Unlike C and C++, there is no implicit equivalence between an array and a pointer to its initial element.

Listing 84: `array_bounds_example.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Array_Bounds_Example is
4   type My_Int is range 0 .. 1000;
5
6   type Index is range 11 .. 15;
7   --           ^ Low bound can
8   --           be any value
9
10  type My_Int_Array is
11    array (Index) of My_Int;
12
13  Tab : constant My_Int_Array :=
14    (2, 3, 5, 7, 11);
15 begin
16   for I in Index loop
17     Put (My_Int'Image (Tab (I)));
18   end loop;
19   New_Line;
20 end Array_Bounds_Example;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Arrays.Array_Bounds_Example`
MD5: `e5fe9e7b83055f3ae23dd890e29c22de`

Runtime output

```
2 3 5 7 11
```

One effect is that the bounds of an array can be any values. In the first example we constructed an array type whose first index is `1`, but in the example above we declare an array type whose first index is `11`.

That's perfectly fine in Ada, and moreover since we use the index type as a range to iterate over the array indices, the code using the array does not need to change.

That leads us to an important consequence with regard to code dealing with arrays. Since the bounds can vary, you should not assume / hard-code specific bounds when iterating / using arrays. That means the code above is good, because it uses the index type, but a for loop as shown below is bad practice even though it works correctly:

```
for I in 11 .. 15 loop
  Tab (I) := Tab (I) * 2;
end loop;
```

Since you can use any discrete type to index an array, enumeration types are permitted.

Listing 85: month_example.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Month_Example is
4   type Month_Duration is range 1 .. 31;
5   type Month is (Jan, Feb, Mar, Apr,
6                 May, Jun, Jul, Aug,
7                 Sep, Oct, Nov, Dec);
8
9   type My_Int_Array is
10    array (Month) of Month_Duration;
11    -- ^ Can use an enumeration type
12    --    as the index
13
14   Tab : constant My_Int_Array :=
15    -- ^ constant is like a variable but
16    --    cannot be modified
17    (31, 28, 31, 30, 31, 30,
18     31, 31, 30, 31, 30, 31);
19    -- Maps months to number of days
20    -- (ignoring leap years)
21
22   Feb_Days : Month_Duration := Tab (Feb);
23   -- Number of days in February
24 begin
25   for M in Month loop
26     Put_Line
27       (Month'Image (M) & " has "
28        & Month_Duration'Image (Tab (M))
29        & " days.");
30     -- ^ Concatenation operator
31   end loop;
32 end Month_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Month_Example
MD5: 420bb8faa36d0efd3d071c76c2033d21

Runtime output

```
JAN has 31 days.
FEB has 28 days.
MAR has 31 days.
```

(continues on next page)

(continued from previous page)

```
APR has 30 days.  
MAY has 31 days.  
JUN has 30 days.  
JUL has 31 days.  
AUG has 31 days.  
SEP has 30 days.  
OCT has 31 days.  
NOV has 30 days.  
DEC has 31 days.
```

In the example above, we are:

- Creating an array type mapping months to month durations in days.
- Creating an array, and instantiating it with an aggregate mapping months to their actual durations in days.
- Iterating over the array, printing out the months, and the number of days for each.

Being able to use enumeration values as indices is very helpful in creating mappings such as shown above one, and is an often used feature in Ada.

7.2 Indexing

We have already seen the syntax for selecting elements of an array. There are however a few more points to note.

First, as is true in general in Ada, the indexing operation is strongly typed. If you use a value of the wrong type to index the array, you will get a compile-time error.

Listing 86: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Greet is  
4   type My_Int is range 0 .. 1000;  
5  
6   type My_Index is range 1 .. 5;  
7   type Your_Index is range 1 .. 5;  
8  
9   type My_Int_Array is  
10    array (My_Index) of My_Int;  
11  
12   Tab : My_Int_Array := (2, 3, 5, 7, 11);  
13 begin  
14   for I in Your_Index loop  
15     Put (My_Int'Image (Tab (I)));  
16     -- ^ Compile time error  
17   end loop;  
18   New_Line;  
19 end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Greet_2
MD5: 54543017e4ec69d24bf9e43d507b50e6

Build output

```
greet.adb:15:31: error: expected type "My_Index" defined at line 6
greet.adb:15:31: error: found type "Your_Index" defined at line 7
gprbuild: *** compilation phase failed
```

Second, arrays in Ada are bounds checked. This means that if you try to access an element outside of the bounds of the array, you will get a run-time error instead of accessing random memory as in unsafe languages.

Listing 87: greet.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4      type My_Int is range 0 .. 1000;
5      type Index is range 1 .. 5;
6
7      type My_Int_Array is
8          array (Index) of My_Int;
9
10     Tab : My_Int_Array := (2, 3, 5, 7, 11);
11 begin
12     for I in Index range 2 .. 6 loop
13         Put (My_Int'Image (Tab (I)));
14         -- ^ Will raise an
15         --      exception when
16         --      I = 6
17     end loop;
18     New_Line;
19 end Greet;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Arrays.Greet_3
MD5: 0102674d089be838f1dfbf0791d99fce
```

Build output

```
greet.adb:12:30: warning: static value out of range of type "Index" defined at
↳ line 5 [enabled by default]
greet.adb:12:30: warning: Constraint_Error will be raised at run time [enabled by
↳ default]
greet.adb:12:30: warning: suspicious loop bound out of range of loop subtype
↳ [enabled by default]
greet.adb:12:30: warning: loop executes zero times or raises Constraint_Error
↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:12 range check failed
```

7.3 Simpler array declarations

In the previous examples, we have always explicitly created an index type for the array. While this can be useful for typing and readability purposes, sometimes you simply want to express a range of values. Ada allows you to do that, too.

Listing 88: simple_array_bounds.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Array_Bounds is
4   type My_Int is range 0 .. 1000;
5
6   type My_Int_Array is
7     array (1 .. 5) of My_Int;
8   --   ^ Subtype of Integer
9
10  Tab : constant My_Int_Array :=
11        (2, 3, 5, 7, 11);
12 begin
13   for I in 1 .. 5 loop
14     --   ^ Subtype of Integer
15     Put (My_Int'Image (Tab (I)));
16   end loop;
17   New_Line;
18 end Simple_Array_Bounds;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Simple_Array_Bounds
MD5: c337a7fe0dacc5f60f7b234aa96d39

Runtime output

```
2 3 5 7 11
```

This example defines the range of the array via the range syntax, which specifies an anonymous subtype of Integer and uses it to index the array.

This means that the type of the index is **Integer**. Similarly, when you use an anonymous range in a for loop as in the example above, the type of the iteration variable is also **Integer**, so you can use I to index Tab.

You can also use a named subtype for the bounds for an array.

7.4 Range attribute

We noted earlier that hard coding bounds when iterating over an array is a bad idea, and showed how to use the array's index type/subtype to iterate over its range in a **for** loop. That raises the question of how to write an iteration when the array has an anonymous range for its bounds, since there is no name to refer to the range. Ada solves that via several attributes of array objects:

Listing 89: range_example.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Range_Example is
4   type My_Int is range 0 .. 1000;
5
6   type My_Int_Array is
7     array (1 .. 5) of My_Int;
8
9   Tab : constant My_Int_Array :=
10        (2, 3, 5, 7, 11);
11 begin

```

(continues on next page)

(continued from previous page)

```

12   for I in Tab'Range loop
13       --      ^ Gets the range of Tab
14       Put (My_Int'Image (Tab (I)));
15   end loop;
16   New_Line;
17 end Range_Example;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Range_Example
MD5: 8b0d7bf346cb59999dfd12dbaaf3e2a6

Runtime output

```
2 3 5 7 11
```

If you want more fine grained control, you can use the separate attributes `'First` and `'Last`.

Listing 90: array_attributes_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Array_Attributes_Example is
4      type My_Int is range 0 .. 1000;
5
6      type My_Int_Array is
7          array (1 .. 5) of My_Int;
8
9      Tab : My_Int_Array :=
10          (2, 3, 5, 7, 11);
11  begin
12      for I in Tab'First .. Tab'Last - 1 loop
13          --      ^ Iterate on every index
14          --      except the last
15          Put (My_Int'Image (Tab (I)));
16      end loop;
17      New_Line;
18  end Array_Attributes_Example;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Array_Attributes_Example
MD5: 95cc407c8aadd936e050fe3505e8fb46

Runtime output

```
2 3 5 7
```

The `'Range`, `'First` and `'Last` attributes in these examples could also have been applied to the array type name, and not just the array instances.

Although not illustrated in the above examples, another useful attribute for an array instance `A` is `A'Length`, which is the number of elements that `A` contains.

It is legal and sometimes useful to have a "null array", which contains no elements. To get this effect, define an index range whose upper bound is less than the lower bound.

7.5 Unconstrained arrays

Let's now consider one of the most powerful aspects of Ada's array facility.

Every array type we have defined so far has a fixed size: every instance of this type will have the same bounds and therefore the same number of elements and the same size.

However, Ada also allows you to declare array types whose bounds are not fixed: in that case, the bounds will need to be provided when creating instances of the type.

Listing 91: unconstrained_array_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Unconstrained_Array_Example is
4      type Days is (Monday, Tuesday, Wednesday,
5                    Thursday, Friday,
6                    Saturday, Sunday);
7
8      type Workload_Type is
9          array (Days range <>) of Natural;
10         -- Indefinite array type
11         --     ^ Bounds are of type Days,
12         --     but not known
13
14     Workload : constant
15         Workload_Type (Monday .. Friday) :=
16         --     ^ Specify the bounds
17         --     when declaring
18         (Friday => 7, others => 8);
19         --     ^ Default value
20         --     ^ Specify element by name of index
21 begin
22     for I in Workload'Range loop
23         Put_Line (Integer'Image (Workload (I)));
24     end loop;
25 end Unconstrained_Array_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Unconstrained_Array_Example
MD5: c84910e9b424cfabbbbe018ba0a6de59

Runtime output

```

8
8
8
8
7
```

The fact that the bounds of the array are not known is indicated by the Days **range** <> syntax. Given a discrete type Discrete_Type, if we use Discrete_Type for the index in an array type then Discrete_Type serves as the type of the index and comprises the range of index values for each array instance.

If we define the index as Discrete_Type **range** <> then Discrete_Type serves as the type of the index, but different array instances may have different bounds from this type.

An array type that is defined with the Discrete_Type **range** <> syntax for its index is referred to as an unconstrained array type, and, as illustrated above, the bounds need to be provided when an instance is created.

The above example also shows other forms of the aggregate syntax. You can specify associations by name, by giving the value of the index on the left side of an arrow association. `1 => 2` thus means "assign value 2 to the element at index 1 in my array". `others => 8` means "assign value 8 to every element that wasn't previously assigned in this aggregate".

⚠ Attention

The so-called "box" notation (`<>`) is commonly used as a wildcard or placeholder in Ada. You will often see it when the meaning is "what is expected here can be anything".

ℹ In other languages

While unconstrained arrays in Ada might seem similar to variable length arrays in C, they are in reality much more powerful, because they're truly first-class values in the language. You can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value. This means that it is useless to pass the bounds or length of an array explicitly along with the array, because they are accessible via the `'First`, `'Last`, `'Range` and `'Length` attributes explained earlier.

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime. This allows Ada to implement unconstrained arrays efficiently; instances can be stored on the stack and do not require heap allocation as in languages like Java.

7.6 Predefined array type: String

A recurring theme in our introduction to Ada types has been the way important built-in types like `Boolean` or `Integer` are defined through the same facilities that are available to the user. This is also true for strings: The `String` type in Ada is a simple array.

Here is how the string type is defined in Ada:

```
type String is
  array (Positive range <>) of Character;
```

The only built-in feature Ada adds to make strings more ergonomic is custom literals, as we can see in the example below.

💡 Hint

String literals are a syntactic sugar for aggregates, so that in the following example, A and B have the same value.

Listing 92: string_literals.ads

```
1 package String_Literals is
2   -- Those two declarations are equivalent
3   A : String (1 .. 11) := "Hello World";
4   B : String (1 .. 11) :=
5       ('H', 'e', 'l', 'l', 'o', ' ',
6        'W', 'o', 'r', 'l', 'd');
7 end String_Literals;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.String_Literals
MD5: 8e5871c8ead4ff8da643539857e23b30

Listing 93: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   Message : String (1 .. 11) := "dlroW olleH";
5   --      ^ Pre-defined array type.
6   --      Component type is Character
7 begin
8   for I in reverse Message'Range loop
9     --      ^ Iterate in reverse order
10    Put (Message (I));
11  end loop;
12  New_Line;
13 end Greet;
```

However, specifying the bounds of the object explicitly is a bit of a hassle; you have to manually count the number of characters in the literal. Fortunately, Ada gives you an easier way.

You can omit the bounds when creating an instance of an unconstrained array type if you supply an initialization, since the bounds can be deduced from the initialization expression.

Listing 94: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   Message : constant String := "dlroW olleH";
5   --      ^ Bounds are automatically
6   --      computed from
7   --      initialization value
8 begin
9   for I in reverse Message'Range loop
10    Put (Message (I));
11  end loop;
12  New_Line;
13 end Greet;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Greet_5
MD5: 21448a1007a07ec9d434880628625c3f

Runtime output

Hello World

Listing 95: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Integer_Array is
5     array (Natural range <>) of Integer;
6
7   My_Array : constant Integer_Array :=
```

(continues on next page)

(continued from previous page)

```

8      (1, 2, 3, 4);
9      ~~~~~
10     --      Bounds are automatically
11     --      computed from
12     --      initialization value
13 begin
14     null;
15 end Main;

```

⚠ Attention

As you can see above, the standard **String** type in Ada is an array. As such, it shares the advantages and drawbacks of arrays: a **String** value is stack allocated, it is accessed efficiently, and its bounds are immutable.

If you want something akin to C++'s `std::string`, you can use *Unbounded Strings* (page 249) from Ada's standard library. This type is more like a mutable, automatically managed string buffer to which you can add content.

7.7 Restrictions

A very important point about arrays: bounds *have* to be known when instances are created. It is for example illegal to do the following.

```

declare
  A : String;
begin
  A := "World";
end;

```

Also, while you of course can change the values of elements in an array, you cannot change the array's bounds (and therefore its size) after it has been initialized. So this is also illegal:

```

declare
  A : String := "Hello";
begin
  A := "World";      -- OK: Same size
  A := "Hello World"; -- Not OK: Different size
end;

```

Also, while you can expect a warning for this kind of error in very simple cases like this one, it is impossible for a compiler to know in the general case if you are assigning a value of the correct length, so this violation will generally result in a run-time error.

i Attention

While we will learn more about this later, it is important to know that arrays are not the only types whose instances might be of unknown size at compile-time.

Such objects are said to be of an *indefinite subtype*, which means that the subtype size is not known at compile time, but is dynamically computed (at run time).

Listing 96: indefinite_subtypes.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Indefinite_Subtypes is
4      function Get_Number return Integer is
5          begin
6              return Integer'Value (Get_Line);
7          end Get_Number;
8
9      A : String := "Hello";
10     -- Indefinite subtype
11
12     B : String (1 .. 5) := "Hello";
13     -- Definite subtype
14
15     C : String (1 .. Get_Number);
16     -- Indefinite subtype
17     -- (Get_Number's value is computed at
18     -- run-time)
19 begin
20     null;
21 end Indefinite_Subtypes;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Indefinite_Subtypes
MD5: a24235838511a94879f74757421a28f0

Here, the `'Value` attribute converts the string to an integer.

7.8 Returning unconstrained arrays

The return type of a function can be any type; a function can return a value whose size is unknown at compile time. Likewise, the parameters can be of any type.

For example, this is a function that returns an unconstrained **String**:

Listing 97: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      type Days is (Monday, Tuesday, Wednesday,
6                   Thursday, Friday,
7                   Saturday, Sunday);
8
9      function Get_Day_Name (Day : Days := Monday)
10         return String is
11      begin
12          return
13              (case Day is
14                  when Monday    => "Monday",
15                  when Tuesday   => "Tuesday",
16                  when Wednesday => "Wednesday",
17                  when Thursday  => "Thursday",
18                  when Friday    => "Friday",
19                  when Saturday  => "Saturday",
20                  when Sunday    => "Sunday");
21      end Get_Day_Name;

```

(continues on next page)

(continued from previous page)

```

22
23 begin
24   Put_Line ("First day is "
25           & Get_Day_Name (Days'First));
26 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Day_Name_1
MD5: 0b7c567c723ded52d8e95c4ef46bcecc

Runtime output

First day is Monday

(This example is for illustrative purposes only. There is a built-in mechanism, the `'Image` attribute for scalar types, that returns the name (as a **String**) of any element of an enumeration type. For example `Days'Image(Monday)` is `"MONDAY"`.)

In other languages

Returning variable size objects in languages lacking a garbage collector is a bit complicated implementation-wise, which is why C and C++ don't allow it, preferring to depend on explicit dynamic allocation / free from the user.

The problem is that explicit storage management is unsafe as soon as you want to collect unused memory. Ada's ability to return variable size objects will remove one use case for dynamic allocation, and hence, remove one potential source of bugs from your programs.

Rust follows the C/C++ model, but with safe pointer semantics. However, dynamic allocation is still used. Ada can benefit from a possible performance edge because it can use any model.

7.9 Declaring arrays (2)

While we can have array types whose size and bounds are determined at run time, the array's component type needs to be of a definite and constrained type.

Thus, if you need to declare, for example, an array of strings, the **String** subtype used as component will need to have a fixed size.

Listing 98: show_days.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Days is
4   type Days is (Monday, Tuesday, Wednesday,
5               Thursday, Friday,
6               Saturday, Sunday);
7
8   subtype Day_Name is String (1 .. 2);
9   -- Subtype of string with known size
10
11  type Days_Name_Type is
12    array (Days) of Day_Name;
13    --      ^ Type of the index
14    --      ^ Type of the element.
15    --      Must be definite

```

(continues on next page)

(continued from previous page)

```
16
17   Names : constant Days_Name_Type :=
18     ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
19   -- Initial value given by aggregate
20 begin
21   for I in Names'Range loop
22     Put_Line (Names (I));
23   end loop;
24 end Show_Days;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Day_Name_2
MD5: bc66303091c084f66abde72ae59f55a9

Runtime output

```
Mo
Tu
We
Th
Fr
Sa
Su
```

7.10 Array slices

One last feature of Ada arrays that we're going to cover is array slices. It is possible to take and use a slice of an array (a contiguous sequence of elements) as a name or a value.

Listing 99: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4    Buf : String := "Hello ...";
5
6    Full_Name : String := "John Smith";
7  begin
8    Buf (7 .. 9) := "Bob";
9    -- Careful! This works because the string
10   -- on the right side is the same length as
11   -- the replaced slice!
12
13   -- Prints "Hello Bob"
14   Put_Line (Buf);
15
16   -- Prints "Hi John"
17   Put_Line ("Hi " & Full_Name (1 .. 4));
18 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Slices
MD5: cdf582c6c9089658236f5c79b7be4c3f

Runtime output

```
Hello Bob
Hi John
```

As we can see above, you can use a slice on the left side of an assignment, to replace only part of an array.

A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice.

⚠ Attention

Ada has [multidimensional arrays](#)¹¹, which are not covered in this course. Slices will only work on one dimensional arrays.

7.11 Renaming

So far, we've seen that the following elements can be renamed: *subprograms* (page 31), *packages* (page 45), and *record components* (page 67). We can also rename objects by using the **renames** keyword. This allows for creating alternative names for these objects. Let's look at an example:

Listing 100: measurements.adb

```
1 package Measurements is
2
3     subtype Degree_Celsius is Float;
4
5     Current_Temperature : Degree_Celsius;
6
7 end Measurements;
```

Listing 101: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Measurements;
3
4 procedure Main is
5     subtype Degrees is
6         Measurements.Degree_Celsius;
7
8     T : Degrees
9         renames Measurements.Current_Temperature;
10 begin
11     T := 5.0;
12
13     Put_Line (Degrees'Image (T));
14     Put_Line (Degrees'Image
15         (Measurements.Current_Temperature));
16
17     T := T + 2.5;
18
19     Put_Line (Degrees'Image (T));
20     Put_Line (Degrees'Image
21         (Measurements.Current_Temperature));
22 end Main;
```

Code block metadata

¹¹ <http://www.ada-auth.org/standards/12rm/html/RM-3-6.html>

Project: Courses.Intro_To_Ada.Arrays.Variable_Renaming
MD5: 4426aeaa364cb5cf10ff40e1bccb9757

Runtime output

```
5.00000E+00
5.00000E+00
7.50000E+00
7.50000E+00
```

In the example above, we declare a variable `T` by renaming the `Current_Temperature` object from the `Measurements` package. As you can see by running this example, both `Current_Temperature` and its alternative name `T` have the same values:

- first, they show the value 5.0
- after the addition, they show the value 7.5.

This is because they are essentially referring to the same object, but with two different names.

Note that, in the example above, we're using `Degrees` as an alias of `Degree_Celsius`. We discussed this method *earlier in the course* (page 62).

Renaming can be useful for improving the readability of more complicated array indexing. Instead of explicitly using indices every time we're accessing certain positions of the array, we can create shorter names for these positions by renaming them. Let's look at the following example:

Listing 102: colors.ads

```
1 package Colors is
2
3     type Color is (Black,
4                   Red,
5                   Green,
6                   Blue,
7                   White);
8
9     type Color_Array is
10        array (Positive range <>) of Color;
11
12     procedure Reverse_It (X : in out Color_Array);
13
14 end Colors;
```

Listing 103: colors.adb

```
1 package body Colors is
2
3     procedure Reverse_It (X : in out Color_Array)
4     is
5     begin
6         for I in X'First ..
7             (X'Last + X'First) / 2
8         loop
9             declare
10                 Tmp      : Color;
11                 X_Left    : Color
12                 renames X (I);
13                 X_Right   : Color
14                 renames X (X'Last + X'First - I);
15             begin
```

(continues on next page)

(continued from previous page)

```

16         Tmp      := X_Left;
17         X_Left   := X_Right;
18         X_Right  := Tmp;
19     end;
20 end loop;
21 end Reverse_It;
22
23 end Colors;

```

Listing 104: test_reverse_colors.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Colors; use Colors;
4
5  procedure Test_Reverse_Colors is
6
7      My_Colors : Color_Array (1 .. 5) :=
8          (Black, Red, Green, Blue, White);
9
10 begin
11     for C of My_Colors loop
12         Put_Line ("My_Color: "
13                 & Color'Image (C));
14     end loop;
15
16     New_Line;
17     Put_Line ("Reversing My_Color...");
18     New_Line;
19     Reverse_It (My_Colors);
20
21     for C of My_Colors loop
22         Put_Line ("My_Color: "
23                 & Color'Image (C));
24     end loop;
25
26 end Test_Reverse_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Arrays.Reverse_Colors
MD5: cd9fd7f64d1ec8967e340d57fd7afc0a

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

```

In the example above, package `Colors` implements the procedure `Reverse_It` by declaring new names for two positions of the array. The actual implementation becomes easy to read:

```
begin
  Tmp      := X_Left;
  X_Left   := X_Right;
  X_Right  := Tmp;
end;
```

Compare this to the alternative version without renaming:

```
begin
  Tmp      := X (I);
  X (I)    := X (X'Last +
                X'First - I);
  X (X'Last + X'First - I) := Tmp;
end;
```

MORE ABOUT TYPES

8.1 Aggregates: A primer

So far, we have talked about aggregates quite a bit and have seen a number of examples. Now we will revisit this feature in some more detail.

An Ada aggregate is, in effect, a literal value for a composite type. It's a very powerful notation that helps you to avoid writing procedural code for the initialization of your data structures in many cases.

A basic rule when writing aggregates is that *every component* of the array or record has to be specified, even components that have a default value.

This means that the following code is incorrect:

Listing 105: incorrect.ads

```
1 package Incorrect is
2   type Point is record
3     X, Y : Integer := 0;
4   end record;
5
6   Origin : Point := (X => 0);
7 end Incorrect;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Incorrect_Aggregate
MD5: 80a3475dece1c42cfb67b1d57b5bd464

Build output

```
incorrect.ads:6:22: error: no value supplied for component "Y"
gprbuild: *** compilation phase failed
```

There are a few shortcuts that you can use to make the notation more convenient:

- To specify the default value for a component, you can use the `<=>` notation.
- You can use the `|` symbol to give several components the same value.
- You can use the **others** choice to refer to every component that has not yet been specified, provided all those fields have the same type.
- You can use the range notation `..` to refer to specify a contiguous sequence of indices in an array.

However, note that as soon as you used a named association, all subsequent components likewise need to be specified with named associations.

Listing 106: points.ads

```

1 package Points is
2   type Point is record
3     X, Y : Integer := 0;
4   end record;
5
6   type Point_Array is
7     array (Positive range <>) of Point;
8
9   -- use the default values
10  Origin : Point := (X | Y => <>);
11
12  -- likewise, use the defaults
13  Origin_2 : Point := (others => <>);
14
15  Points_1 : Point_Array := ((1, 2), (3, 4));
16  Points_2 : Point_Array := (1      => (1, 2),
17                             2      => (3, 4),
18                             3 .. 20 => <>);
19 end Points;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Points
MD5: 48ea183a42f203325ed6190fbd8493d9

8.2 Overloading and qualified expressions

Ada has a general concept of name overloading, which we saw earlier in the section on *enumeration types* (page 51).

Let's take a simple example: it is possible in Ada to have functions that have the same name, but different types for their parameters.

Listing 107: pkg.ads

```

1 package Pkg is
2   function F (A : Integer) return Integer;
3   function F (A : Character) return Integer;
4 end Pkg;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Overloading
MD5: defae85228ee183b536af395d077e71e

This is a common concept in programming languages, called *overloading*¹², or name overloading.

One of the novel aspects of Ada's overloading facility is the ability to resolve overloading based on the return type of a function.

Listing 108: pkg.ads

```

1 package Pkg is
2   type SSID is new Integer;
3
```

(continues on next page)

¹² https://en.wikipedia.org/wiki/Function_overloading

(continued from previous page)

```

4     function Convert (Self : SSID)
5         return Integer;
6     function Convert (Self : SSID)
7         return String;
8 end Pkg;

```

Listing 109: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Pkg;         use Pkg;
3
4 procedure Main is
5     S : String := Convert (123_145_299);
6     --           ^ Valid, will choose the
7     --           proper Convert
8 begin
9     Put_Line (S);
10 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Overloading
MD5: aa556b55ee89f9c5f8f7e138d84c27b8

⚠ Attention

Note that overload resolution based on the type is allowed for both functions and enumeration literals in Ada - which is why you can have multiple enumeration literals with the same name. Semantically, an enumeration literal is treated like a function that has no parameters.

However, sometimes an ambiguity makes it impossible to resolve which declaration of an overloaded name a given occurrence of the name refers to. This is where a qualified expression becomes useful.

Listing 110: pkg.ads

```

1 package Pkg is
2     type SSID is new Integer;
3
4     function Convert (Self : SSID)
5         return Integer;
6     function Convert (Self : SSID)
7         return String;
8     function Convert (Self : Integer)
9         return String;
10 end Pkg;

```

Listing 111: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Pkg;         use Pkg;
3
4 procedure Main is
5     S : String := Convert (123_145_299);
6     --           ^ Invalid, which convert
7     --           should we call?
8

```

(continues on next page)

(continued from previous page)

```

9   S2 : String := Convert (SSID'(123_145_299));
10  --           ^ We specify that the
11  --           type of the
12  --           expression is SSID.
13
14  -- We could also have declared a temporary
15
16  I : SSID := 123_145_299;
17
18  S3 : String := Convert (I);
19  begin
20      Put_Line (S);
21  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Overloading_Error
MD5: 722660d8b692cde65a1c2b7800dd78c4

Syntactically the target of a qualified expression can be either any expression in parentheses, or an aggregate:

Listing 112: qual_expr.ads

```

1  package Qual_Expr is
2      type Point is record
3          A, B : Integer;
4      end record;
5
6      P : Point := Point'(12, 15);
7
8      A : Integer := Integer'(12);
9  end Qual_Expr;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Qual_Expr
MD5: e71523eb441a28a4f6549d5f0418620a

This illustrates that qualified expressions are a convenient (and sometimes necessary) way for the programmer to make the type of an expression explicit, for the compiler of course, but also for other programmers.

⚠ Attention

While they look and feel similar, type conversions and qualified expressions are *not* the same.

A qualified expression specifies the exact type that the target expression will be resolved to, whereas a type conversion will try to convert the target and issue a run-time error if the target value cannot be so converted.

Note that you can use a qualified expression to convert from one subtype to another, with an exception raised if a constraint is violated.

```
X : Integer := Natural'(1);
```

8.3 Character types

As noted earlier, each enumeration type is distinct and incompatible with every other enumeration type. However, what we did not mention previously is that character literals are permitted as enumeration literals. This means that in addition to the language's strongly typed character types, user-defined character types are also permitted:

Listing 113: character_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Character_Example is
4      type My_Char is ('a', 'b', 'c');
5      -- Our custom character type, an
6      -- enumeration type with 3 valid values.
7
8      C : Character;
9      -- ^ Built-in character type
10     -- (it's an enumeration type)
11
12     M : My_Char;
13 begin
14     C := '?';
15     -- ^ Character literal
16     -- (enumeration literal)
17
18     M := 'a';
19
20     C := 65;
21     -- ^ Invalid: 65 is not a
22     -- Character value
23
24     C := Character'Val (65);
25     -- Assign the character at
26     -- position 65 in the
27     -- enumeration (which is 'A')
28
29     M := C;
30     -- ^ Invalid: C is of type Character,
31     -- and M is a My_Char
32
33     M := 'd';
34     -- ^ Invalid: 'd' is not a valid
35     -- literal for type My_Char
36 end Character_Example;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Types.Character_Example
MD5: e4c5a07dbe8745749056f8c110d69fa3

Build output

```

character_example.adb:20:09: error: expected type "Standard.Character"
character_example.adb:20:09: error: found type universal integer
character_example.adb:29:09: error: expected type "My_Char" defined at line 4
character_example.adb:29:09: error: found type "Standard.Character"
character_example.adb:33:09: error: character not defined for type "My_Char"
↳ defined at line 4
gprbuild: *** compilation phase failed

```

In this example, we're using characters in the definition of My_Char.

ACCESS TYPES (POINTERS)

9.1 Overview

Pointers are a potentially dangerous construct, which conflicts with Ada's underlying philosophy.

There are two ways in which Ada helps shield programmers from the dangers of pointers:

1. One approach, which we have already seen, is to provide alternative features so that the programmer does not need to use pointers. Parameter modes, arrays, and varying size types are all constructs that can replace typical pointer usages in C.
2. Second, Ada has made pointers as safe and restricted as possible, but allows "escape hatches" when the programmer explicitly requests them and presumably will be exercising such features with appropriate care.

Here is how you declare a simple pointer type, or access type, in Ada:

Listing 114: dates.ads

```
1 package Dates is
2   type Months is
3     (January, February, March, April,
4      May, June, July, August, September,
5      October, November, December);
6
7   type Date is record
8     Day   : Integer range 1 .. 31;
9     Month : Months;
10    Year  : Integer;
11  end record;
12 end Dates;
```

Listing 115: access_types.ads

```
1 with Dates; use Dates;
2
3 package Access_Types is
4   -- Declare an access type
5   type Date_Acc is access Date;
6   --           ^ "Designated type"
7   --           ^ Date_Acc values
8   --           point to Date
9   --           objects
10
11   D : Date_Acc := null;
12   --           ^ Literal for
13   --           "access to nothing"
```

(continues on next page)

(continued from previous page)

```
14  -- ^ Access to date
15  end Access_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: d3421918c48c221836bdf03b9e68bfb5

This illustrates how to:

- Declare an access type whose values point to ("designate") objects from a specific type
- Declare a variable (access value) from this access type
- Give it a value of **null**

In line with Ada's strong typing philosophy, if you declare a second access type whose designated type is Date, the two access types will be incompatible with each other:

Listing 116: access_types.ads

```
1  with Dates; use Dates;
2
3  package Access_Types is
4      -- Declare an access type
5      type Date_Acc is access Date;
6      type Date_Acc_2 is access Date;
7
8      D : Date_Acc := null;
9      D2 : Date_Acc_2 := D;
10     -- ^ Invalid! Different types
11 end Access_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: af0dff5a26cb16f0fe15c84286557a44

Build output

```
access_types.ads:9:24: error: expected type "Date_Acc_2" defined at line 6
access_types.ads:9:24: error: found type "Date_Acc" defined at line 5
gprbuild: *** compilation phase failed
```

In other languages

In most other languages, pointer types are structurally, not nominally typed, like they are in Ada, which means that two pointer types will be the same as long as they share the same target type and accessibility rules.

Not so in Ada, which takes some time getting used to. A seemingly simple problem is, if you want to have a canonical access to a type, where should it be declared? A commonly used pattern is that if you need an access type to a specific type you "own", you will declare it along with the type:

```
package Access_Types is
    type Point is record
        X, Y : Natural;
    end record;

    type Point_Access is access Point;
end Access_Types;
```

9.2 Allocation (by type)

Once we have declared an access type, we need a way to give variables of the types a meaningful value! You can allocate a value of an access type with the **new** keyword in Ada.

Listing 117: access_types.ads

```

1 with Dates; use Dates;
2
3 package Access_Types is
4     type Date_Acc is access Date;
5
6     D : Date_Acc := new Date;
7     --           ^ Allocate a new Date record
8 end Access_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: e0be95b966e4aebaaf25db646d60c35c

If the type you want to allocate needs constraints, you can put them in the subtype indication, just as you would do in a variable declaration:

Listing 118: access_types.ads

```

1 with Dates; use Dates;
2
3 package Access_Types is
4     type String_Acc is access String;
5     --           ^
6     -- Access to unconstrained array type
7     Msg : String_Acc;
8     --     ^ Default value is null
9
10    Buffer : String_Acc :=
11        new String (1 .. 10);
12    --           ^ Constraint required
13 end Access_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: 83cf7a1074ff1b739658508098aa8208

In some cases, though, allocating just by specifying the type is not ideal, so Ada also allows you to initialize along with the allocation. This is done via the qualified expression syntax:

Listing 119: access_types.ads

```

1 with Dates; use Dates;
2
3 package Access_Types is
4     type Date_Acc is access Date;
5     type String_Acc is access String;
6
7     D : Date_Acc :=
8         new Date'(30, November, 2011);
```

(continues on next page)

(continued from previous page)

```

9   Msg : String_Acc := new String'("Hello");
10  end Access_Types;

```

9.3 Dereferencing

The last important piece of Ada's access type facility is how to get from an access value to the object that is pointed to, that is, how to dereference the pointer. Dereferencing a pointer uses the `.all` syntax in Ada, but is often not needed — in many cases, the access value will be implicitly dereferenced for you:

Listing 120: access_types.ads

```

1  with Dates; use Dates;
2
3  package Access_Types is
4      type Date_Acc is access Date;
5
6      D      : Date_Acc :=
7              new Date'(30, November, 2011);
8
9      Today : Date := D.all;
10     --      ^ Access value dereference
11     J      : Integer := D.Day;
12     --      ^ Implicit dereference
13     --      for record and array
14     --      components
15     --      Equivalent to D.all.day
16  end Access_Types;

```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Access_Types
MD5: 5cd1c259da04010b0dc1b43e9bd93b55

9.4 Other features

As you might know if you have used pointers in C or C++, we are still missing features that are considered fundamental to the use of pointers, such as:

- Pointer arithmetic (being able to increment or decrement a pointer in order to point to the next or previous object)
- Manual deallocation - what is called `free` or `delete` in C. This is a potentially unsafe operation. To keep within the realm of safe Ada, you need to never deallocate manually.

Those features exist in Ada, but are only available through specific standard library APIs.

Attention

The guideline in Ada is that most of the time you can avoid manual allocation, and you should.

There are many ways to avoid manual allocation, some of which have been covered (such as parameter modes). The language also provides library abstractions to avoid pointers:

1. One is the use of *containers* (page 205). Containers help users avoid pointers, because container memory is automatically managed.
2. A container to note in this context is the *Indefinite holder*¹³. This container allows you to store a value of an indefinite type such as String.
3. GNATCOLL has a library for smart pointers, called *Refcount*¹⁴. Those pointers' memory is automatically managed, so that when an allocated object has no more references to it, the memory is automatically deallocated.

9.5 Mutually recursive types

The linked list is a common idiom in data structures; in Ada this would be most naturally defined through two types, a record type and an access type, that are mutually dependent. To declare mutually dependent types, you can use an incomplete type declaration:

Listing 121: simple_list.ads

```

1 package Simple_List is
2   type Node;
3   -- This is an incomplete type declaration,
4   -- which is completed in the same
5   -- declarative region.
6
7   type Node_Acc is access Node;
8
9   type Node is record
10     Content    : Natural;
11     Prev, Next : Node_Acc;
12   end record;
13 end Simple_List;
```

Code block metadata

Project: Courses.Intro_To_Ada.Access_Types.Simple_List
MD5: 4929b89c1fc913da635fa02e48248271

In this example, the Node and Node_Acc types are mutually dependent.

¹³ <http://www.ada-auth.org/standards/12rat/html/Rat12-8-5.html>

¹⁴ <https://github.com/AdaCore/gnatcoll-core/blob/master/src/gnatcoll-refcount.ads>

MORE ABOUT RECORDS

10.1 Dynamically sized record types

We have previously seen *some simple examples of record types* (page 65). Let's now look at some of the more advanced properties of this fundamental language feature.

One point to note is that object size for a record type does not need to be known at compile time. This is illustrated in the example below:

Listing 122: runtime_length.ads

```
1 package Runtime_Length is
2   function Compute_Max_Len return Natural;
3 end Runtime_Length;
```

Listing 123: var_size_record.ads

```
1 with Runtime_Length; use Runtime_Length;
2
3 package Var_Size_Record is
4   Max_Len : constant Natural :=
5       Compute_Max_Len;
6   --      ^ Not known at compile time
7
8   type Items_Array is
9       array (Positive range <>) of Integer;
10
11   type Growable_Stack is record
12       Items : Items_Array (1 .. Max_Len);
13       Len   : Natural;
14   end record;
15   -- Growable_Stack is a definite type, but
16   -- size is not known at compile time.
17
18   G : Growable_Stack;
19 end Var_Size_Record;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Var_Size_Record
MD5: 6fb0b3f2b685a72ec694640ce378f77c

It is completely fine to determine the size of your records at run time, but note that all objects of this type will have the same size.

10.2 Records with discriminant

In the example above, the size of the `Items` field is determined once, at run-time, but every `Growable_Stack` instance will be exactly the same size. But maybe that's not what you want to do. We saw that arrays in general offer this flexibility: for an unconstrained array type, different objects can have different sizes.

You can get analogous functionality for records, too, using a special kind of field that is called a discriminant:

Listing 124: `var_size_record_2.ads`

```

1 package Var_Size_Record_2 is
2   type Items_Array is
3     array (Positive range <>) of Integer;
4
5   type Growable_Stack (Max_Len : Natural) is
6     record
7       --           ^ Discriminant. Cannot be
8       --           modified once
9       --           initialized.
10      Items : Items_Array (1 .. Max_Len);
11      Len   : Natural := 0;
12    end record;
13    -- Growable_Stack is an indefinite type
14    -- (like an array)
15 end Var_Size_Record_2;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Var_Size_Record_2
MD5: 0c2ffe41b7553984e1ef48a50386559f

Discriminants, in their simple forms, are constant: You cannot modify them once you have initialized the object. This intuitively makes sense since they determine the size of the object.

Also, they make a type indefinite: Whether or not the discriminant is used to specify the size of an object, a type with a discriminant will be indefinite if the discriminant is not declared with an initialization:

Listing 125: `test_discriminants.ads`

```

1 package Test_Discriminants is
2   type Point (X, Y : Natural) is record
3     null;
4   end record;
5
6   P : Point;
7   -- ERROR: Point is indefinite, so you
8   -- need to specify the discriminants
9   -- or give a default value
10
11   P2 : Point (1, 2);
12   P3 : Point := (1, 2);
13   -- Those two declarations are equivalent.
14
15 end Test_Discriminants;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Test_Discriminants
 MD5: c3ec81ccae0d4144fe952ad99482be81

Build output

```
test_discriminants.ads:6:08: error: unconstrained subtype not allowed (need ↵
↵ initialization)
test_discriminants.ads:6:08: error: provide initial value or explicit discriminant ↵
↵ values
test_discriminants.ads:6:08: error: or give default discriminant values for type
↵ "Point"
gprbuild: *** compilation phase failed
```

This also means that, in the example above, you cannot declare an array of Point values, because the size of a Point is not known.

As mentioned in the example above, we could provide a default value for the discriminants, so that we could legally declare Point values without specifying the discriminants. For the example above, this is how it would look:

Listing 126: test_discriminants.ads

```
1 package Test_Discriminants is
2   type Point (X, Y : Natural := 0) is record
3     null;
4   end record;
5
6   P : Point;
7   -- We can now simply declare a "Point"
8   -- without further ado. In this case,
9   -- we're using the default values (0)
10  -- for X and Y.
11
12  P2 : Point (1, 2);
13  P3 : Point := (1, 2);
14  -- We can still specify discriminants.
15
16 end Test_Discriminants;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Test_Discriminants
 MD5: 259f6cdf7fa857cc006dac6d1daedd73

Also note that, even though the Point type now has default discriminants, we can still specify discriminants, as we're doing in the declarations of P2 and P3.

In most other respects discriminants behave like regular fields: You have to specify their values in aggregates, as seen above, and you can access their values via the dot notation.

Listing 127: main.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Var_Size_Record_2; use Var_Size_Record_2;
4
5 procedure Main is
6   procedure Print_Stack (G : Growable_Stack) is
7     begin
8       Put ("<Stack, items: [");
9       for I in G.Items'Range loop
10        exit when I > G.Len;
```

(continues on next page)

(continued from previous page)

```

11     Put (" " & Integer'Image (G.Items (I)));
12   end loop;
13   Put_Line ("]>");
14 end Print_Stack;
15
16 S : Growable_Stack :=
17   (Max_Len => 128,
18    Items  => (1, 2, 3, 4, others => <>),
19    Len    => 4);
20 begin
21   Print_Stack (S);
22 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Var_Size_Record_2
MD5: 4e8c102cd93dc5d8aa1b402589c5239b

Runtime output

```
<Stack, items: [ 1 2 3 4]>
```

Note

In the examples above, we used a discriminant to determine the size of an array, but it is not limited to that, and could be used, for example, to determine the size of a nested discriminated record.

10.3 Variant records

The examples of discriminants thus far have illustrated the declaration of records of varying size, by having components whose size depends on the discriminant.

However, discriminants can also be used to obtain the functionality of what are sometimes called "variant records": records that can contain different sets of fields.

Listing 128: variant_record.ads

```

1 package Variant_Record is
2   -- Forward declaration of Expr
3   type Expr;
4
5   -- Access to a Expr
6   type Expr_Access is access Expr;
7
8   type Expr_Kind_Type is (Bin_Op_Plus,
9                           Bin_Op_Minus,
10                          Num);
11   -- A regular enumeration type
12
13   type Expr (Kind : Expr_Kind_Type) is record
14     -- ^ The discriminant is an
15     -- enumeration value
16     case Kind is
17       when Bin_Op_Plus | Bin_Op_Minus =>
18         Left, Right : Expr_Access;
19       when Num =>

```

(continues on next page)

(continued from previous page)

```

20         Val : Integer;
21     end case;
22     -- Variant part. Only one, at the end of
23     -- the record definition, but can be
24     -- nested
25 end record;
26 end Variant_Record;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Variant_Record
MD5: af9c1edca3ed6b2d938249c7258806b1

The fields that are in a **when** branch will be only available when the value of the discriminant is covered by the branch. In the example above, you will only be able to access the fields Left and Right when the Kind is Bin_Op_Plus or Bin_Op_Minus.

If you try to access a field that is not valid for your record, a Constraint_Error will be raised.

Listing 129: main.adb

```

1  with Variant_Record; use Variant_Record;
2
3  procedure Main is
4      E : Expr := (Num, 12);
5  begin
6      E.Left := new Expr'(Num, 15);
7      -- Will compile but fail at runtime
8  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Variant_Record
MD5: d157d5f96db0825b9376ba7fca9613ed

Build output

```

main.adb:6:05: warning: component not present in subtype of "Expr" defined at line_
↪4 [enabled by default]
main.adb:6:05: warning: Constraint_Error will be raised at run time [enabled by_
↪default]

```

Runtime output

```

raised CONSTRAINT_ERROR : main.adb:6 discriminant check failed

```

Here is how you could write an evaluator for expressions:

Listing 130: main.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2
3  with Variant_Record; use Variant_Record;
4
5  procedure Main is
6      function Eval_Expr (E : Expr) return Integer is
7          (case E.Kind is
8              when Bin_Op_Plus =>
9                  Eval_Expr (E.Left.all)

```

(continues on next page)

(continued from previous page)

```
10         + Eval_Expr (E.Right.all),
11     when Bin_Op_Minus =>
12         Eval_Expr (E.Left.all)
13         - Eval_Expr (E.Right.all),
14     when Num => E.Val);
15
16     E : Expr := (Bin_Op_Plus,
17                 new Expr'(Bin_Op_Minus,
18                           new Expr'(Num, 12),
19                           new Expr'(Num, 15)),
20                 new Expr'(Num, 3));
21 begin
22     Put_Line (Integer'Image (Eval_Expr (E)));
23 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.More_About_Records.Variant_Record
MD5: 807dbb921b44b3eaeaf1baf6ffef1afaa

Runtime output

0

In other languages

Ada's variant records are very similar to Sum types in functional languages such as OCaml or Haskell. A major difference is that the discriminant is a separate field in Ada, whereas the 'tag' of a Sum type is kind of built in, and only accessible with pattern matching.

There are other differences (you can have several discriminants in a variant record in Ada). Nevertheless, they allow the same kind of type modeling as sum types in functional languages.

Compared to C/C++ unions, Ada variant records are more powerful in what they allow, and are also checked at run time, which makes them safer.

FIXED-POINT TYPES

In this chapter, we discuss fixed-point types, which can be classified in two categories: *decimal fixed-point types* (page 107) and *ordinary (binary) fixed-point types* (page 111). Afterward a brief overview of each category, we discuss some *differences between fixed-point and floating-point types* (page 112).

11.1 Decimal fixed-point types

We have already seen how to specify *floating-point types* (page 52). However, in some applications floating-point is not appropriate since, for example, the roundoff error from binary arithmetic may be unacceptable or perhaps the hardware does not support floating-point instructions. Ada provides a category of types, the decimal fixed-point types, that allows the programmer to specify the required decimal precision (number of digits) as well as the scaling factor (a power of ten) and, optionally, a range. In effect the values will be represented as integers implicitly scaled by the specified power of 10. This is useful, for example, for financial applications.

The syntax for a simple decimal fixed-point type is

```
type <type-name> is
  delta <delta-value> digits <digits-value>;
```

In this case, the **delta** and the **digits** will be used by the compiler to derive a range.

11.1.1 Decimal delta

The delta determines the required decimal precision for the type. For example, if we want to be able to use two digits after the decimal point, we would write **delta 10.0 ** (-2)** — which is equivalent to **delta 0.01**. (You can use any of those definitions: both **delta 10.0 ** (-2)** and **delta 0.01** are correct.)

Listing 131: decimal_fixed_point_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Fixed_Point_Types is
4   type Decimal is
5     delta 10.0 ** (-1) digits 3;
6
7   -- Alternatively:
8   -- type Decimal is
9   --   delta 0.1 digits 3;
10 begin
11   Put_Line
12     ("The decimal precision of Decimal is "
13      & Decimal'Delta'Image);
14 end Decimal_Fixed_Point_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Fixed_Point_Types
MD5: bcb69d188151413ee4f4a413d85bb0de

Runtime output

The decimal precision of Decimal is 1.0000000000000000E-01

In this example, we declare the `Decimal` type, which has a decimal precision of 0.1. We use the '`Delta`' attribute to show the decimal precision of the type.

11.1.2 Decimal digits

Unsurprisingly, the `digits` part of the type declaration determines the number of digits that a type is able to represent. For example, by writing `digits 3`, we're able to represent values with three digits ranging from -999 to 999. For example:

Listing 132: decimal_fixed_point_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Fixed_Point_Types is
4   type Decimal is
5     delta 10.0 ** (0) digits 3;
6 begin
7   Put_Line ("The minimum value of Decimal is "
8     & Decimal'First'Image);
9   Put_Line ("The maximum value of Decimal is "
10     & Decimal'Last'Image);
11 end Decimal_Fixed_Point_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Fixed_Point_Types
MD5: 8b130008590ca44d7ff588aef1ad3270

Runtime output

The minimum value of Decimal is -999.0
The maximum value of Decimal is 999.0

In this example, we declare the `Decimal` type, which has a range from -999 to 999. We use the '`First`' and '`Last`' attributes to show the first and last value of the range, respectively.

For further reading...

When running the application above, we see that the first and last values are -999.0 and 999.0, respectively — i.e. values with the decimal point. Strictly speaking, however, the actual first and last values are -999 and 999 because we selected a delta of 1.0. The decimal point (.0) we see in the application output (in the values -999.0 and 999.0) is only there to indicate that this is not an integer value — however, it doesn't indicate an extended decimal precision at all.

Listing 133: decimal_fixed_point_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Decimal_Fixed_Point_Types is
4      type Decimal is
5          delta 10.0 ** (0) digits 3;
6
7      D : Decimal := 0.1;
8          --      ^^^
9      --  ERROR: value cannot be represented
10     --      by Decimal type.
11  begin
12      Put_Line (D'Image);
13  end Decimal_Fixed_Point_Types;

```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Fixed_Point_Type_
 ↳ Precision_Error
 MD5: d169d7796b61392baada9795ee8b8187

Build output

```

decimal_fixed_point_types.adb:7:19: error: value has extraneous low order digits
gprbuild: *** compilation phase failed

```

Assigning the value 0.1 to D is wrong because the Decimal type cannot represent this value.

i For further reading...

The `Decimal` type above is similar to — but far from being equivalent to — the following floating-point type declaration:

Listing 134: `decimal_fixed_point_types.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Fixed_Point_Types is
4   type Float_999 is
5     digits 3
6     range -999.0 .. 999.0;
7 begin
8   Put_Line ("The minimum value of Float_999 is "
9             & Float_999'First'Image);
10  Put_Line ("The maximum value of Float_999 is "
11            & Float_999'Last'Image);
12 end Decimal_Fixed_Point_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Floating_Point_Range_Equivalent
MD5: 49a6316e1c6846da75d5d9862b1f7ef8

Runtime output

The minimum value of Float_999 is -9.99000E+02
The maximum value of Float_999 is 9.99000E+02

The `Float_999` type from this example has (roughly) the same range as the `Decimal` type that we declared in the previous example: -999 to 999. However, there are substantial *differences between fixed-point and floating-point types* (page 112), so we cannot say that these type declarations are equivalent.

11.1.3 Decimal delta and digits

By combining those three digits (i.e. `digits 3`) with a decimal precision of two digits after the decimal point (`delta 10.0 ** (-2)`), we get a range from -9.99 to 9.99. For example:

Listing 135: `decimal_fixed_point_types.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Fixed_Point_Types is
4   type Decimal is
5     delta 10.0 ** (-2) digits 3;
6 begin
7   Put_Line ("The minimum value of Decimal is "
8             & Decimal'First'Image);
9   Put_Line ("The maximum value of Decimal is "
10            & Decimal'Last'Image);
11 end Decimal_Fixed_Point_Types;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Fixed_Point_Types
MD5: 2f55fc3d0f7f77c5f17f750680933422

Runtime output

The minimum value of Decimal is -9.99
 The maximum value of Decimal is 9.99

In this example, we declare the Decimal type, which has a range from -9.99 to 9.99 (as expected).

11.1.4 Requirements for the delta

Note that the delta expression for decimal fixed-point types must be a power of 10. Using a different value for the power leads to compilation errors. For example:

Listing 136: decimal_fixed_point_type_error.ads

```

1 package Decimal_Fixed_Point_Type_Error is
2
3   type Decimal_Error_1 is
4     delta 2.0 ** (-1) digits 3;
5     -- ^^^^^^^^^^^
6     -- ERROR: not power of ten
7
8   type Decimal_Error_2 is
9     delta 0.125 digits 3;
10    -- ^^^^^
11    -- ERROR: not power of ten
12
13 end Decimal_Fixed_Point_Type_Error;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Fixed_Point_Type_Error
 MD5: f4541eb0256b3770c1b8a9fd4a6c02fe

Build output

```

decimal_fixed_point_type_error.ads:4:06: error: delta expression must be a power_
↳ of 10
decimal_fixed_point_type_error.ads:9:06: error: delta expression must be a power_
↳ of 10
gprbuild: *** compilation phase failed
```

In this example, the type declarations (of Decimal_Error_1 and Decimal_Error_2) are wrong because the delta expression is not a power of 10.

11.2 Ordinary fixed-point types

Ordinary fixed-point types are similar to decimal fixed-point types in that the values are, in effect, scaled integers. The difference between them is in the scale factor: for a decimal fixed-point type, the scaling, given explicitly by the type's **delta**, is always a power of ten. In contrast, for an ordinary fixed-point type, the **delta** isn't limited to power of 10 values, but it can have any arbitrary base.

For further reading...

When representing ordinary fixed-point types on the machine, the compiler selects a scaling factor derived from the value of **delta** specified in the type declaration. This compiler-selected scaling factor is, by default, a power of two — even if the value provided for the **delta** isn't a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types.

The syntax for an ordinary fixed-point type is

```
type <type-name> is
  delta <delta-value>
  range <lower-bound> .. <upper-bound>;
```

For example, we can define an ordinary fixed-point type `T_Inv_Trig` for inverse trigonometric calculations:

Listing 137: `custom_fixed_point_range.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics; use Ada.Numerics;
3
4 procedure Custom_Fixed_Point_Range is
5   type T_Inv_Trig is
6     delta 0.0005
7     range -Pi / 2.0 .. Pi / 2.0;
8 begin
9   Put_Line ("Delta    value of T_Inv_Trig: "
10            & T_Inv_Trig'Image
11            (T_Inv_Trig'Delta));
12   Put_Line ("Minimum  value of T_Inv_Trig: "
13            & T_Inv_Trig'Image
14            (T_Inv_Trig'First));
15   Put_Line ("Maximum  value of T_Inv_Trig: "
16            & T_Inv_Trig'Image
17            (T_Inv_Trig'Last));
18 end Custom_Fixed_Point_Range;
```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Custom_Fixed_Point_Range
MD5: 41b251e49ab887496ecd491431b7ffb0

Runtime output

```
Delta    value of T_Inv_Trig:  0.0005
Minimum  value of T_Inv_Trig: -1.5708
Maximum  value of T_Inv_Trig:  1.5708
```

In this example, we are defining the `T_Inv_Trig` type with a range from $-\pi/2$ to $\pi/2$, and a delta of 0.0005. Note that, in this case, the delta is neither a power of ten nor a power of two. (In fact, this value corresponds to `2000.0 ** (-1)`.)

11.3 Fixed-point vs. floating-point types

The main difference between fixed-point and floating-point types is that fixed-point types don't have an exponent. This has an impact on calculations using small values: while they might still be representable with floating-point types, those small values might simply *disappear* (i.e. become zero) in the fixed-point representation. Let's see an example where we compare the decimal type `Decimal` to the floating-point type `Float_32`:

Listing 138: `decimal_vs_floating_point_types.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Vs_Floating_Point_Types is
4   type Decimal is
5     delta 10.0 ** (-2) digits 9;
```

(continues on next page)

(continued from previous page)

```

6
7  type Float_32 is
8      digits 6
9      range -9999999.99 .. 9999999.99;
10
11  D : Decimal := 0.01;
12  F : Float_32 := 0.01;
13  begin
14      Put_Line ("D = " &
15              D'Image);
16      Put_Line ("F = " &
17              F'Image);
18
19      D := D / 2.0;
20      -- ^^^^^^
21      -- Value becomes zero.
22
23      F := F / 2.0;
24      -- ^^^^^^
25      -- Exponent is used to
26      -- represent smaller
27      -- value.
28
29      Put_Line ("D = " &
30              D'Image);
31      Put_Line ("F = " &
32              F'Image);
33  end Decimal_Vs_Floating_Point_Types;

```

Code block metadata

Project: Courses.Intro_To_Ada.Fixed_Point_Types.Decimal_Vs_Floating_Point_Types
MD5: dbbe2642fb64ae54c7634274e8ff6c79

Runtime output

```

D = 0.01
F = 1.00000E-02
D = 0.00
F = 5.00000E-03

```

Both types in this example have roughly the same size and range. However, the result of the divide-by-two operation isn't the same: because of the exponent, F has the expected value (0.005) after the operation, while the value of D is zero. The reason is that the resulting value 0.005 cannot be represented by the decimal precision of the `Decimal` type. In the case of F, however, the value can be represented due to a simple change in the exponent.

This lack of precision we just described might seem like a drawback for fixed-point types. However, depending on the algorithm and its field of application, this is the exact behavior that we might be looking for. As mentioned in the beginning of this chapter, financial applications benefit from decimal types, while using floating-point type for these applications can lead to unpredictable (or undesirable) behavior.

Another major difference concerns the way fixed-point operations translate into machine operations. In most cases, operations on fixed-point types are modeled in a processor by using integer registers and instructions. Essentially, the compiler maps fixed-point types to integer types, but it uses slightly different numeric rules. This fact can be an advantage for specific embedded applications where a floating-point unit might be either non-existent or its usage might have a higher associated cost in terms of CPU cycles or power consumption. Therefore, for these specific applications, using fixed-point types could be considered as an alternative.

PRIVACY

One of the main principles of modular programming, as well as object oriented programming, is [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))¹⁵.

Encapsulation, briefly, is the concept that the implementer of a piece of software will distinguish between the code's public interface and its private implementation.

This is not only applicable to software libraries but wherever abstraction is used.

In Ada, the granularity of encapsulation is a bit different from most object-oriented languages, because privacy is generally specified at the package level.

12.1 Basic encapsulation

Listing 139: encapsulate.ads

```
1 package Encapsulate is
2     procedure Hello;
3
4 private
5
6     procedure Hello2;
7     -- Not visible from external units
8 end Encapsulate;
```

Listing 140: encapsulate.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate is
4
5     procedure Hello is
6     begin
7         Put_Line ("Hello");
8     end Hello;
9
10    procedure Hello2 is
11    begin
12        Put_Line ("Hello #2");
13    end Hello2;
14
15 end Encapsulate;
```

¹⁵ [https://en.wikipedia.org/wiki/Encapsulation_\(computer_programming\)](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

Listing 141: main.adb

```
1 with Encapsulate;
2
3 procedure Main is
4 begin
5     Encapsulate.Hello;
6     Encapsulate.Hello2;
7     -- Invalid: Hello2 is not visible
8 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Encapsulate
MD5: cf56ee89481962d1e0a6d1e9ad888362

Build output

main.adb:6:15: error: "Hello2" is not a visible entity of "Encapsulate"
gprbuild: *** compilation phase failed

12.2 Abstract data types

With this high-level granularity, it might not seem obvious how to hide the implementation details of a type. Here is how it can be done in Ada:

Listing 142: stacks.ads

```
1 package Stacks is
2     type Stack is private;
3     -- Declare a private type: You cannot depend
4     -- on its implementation. You can only assign
5     -- and test for equality.
6
7     procedure Push (S : in out Stack;
8                     Val : Integer);
9     procedure Pop (S : in out Stack;
10                  Val : out Integer);
11 private
12
13     subtype Stack_Index is
14         Natural range 1 .. 10;
15
16     type Content_Type is
17         array (Stack_Index) of Natural;
18
19     type Stack is record
20         Top : Stack_Index;
21         Content : Content_Type;
22     end record;
23 end Stacks;
```

Listing 143: stacks.adb

```
1 package body Stacks is
2
3     procedure Push (S : in out Stack;
4                     Val : Integer) is
5     begin
```

(continues on next page)

(continued from previous page)

```

6      -- Missing implementation!
7      null;
8  end Push;
9
10     procedure Pop (S : in out Stack;
11                   Val : out Integer) is
12     begin
13       -- Dummy implementation!
14       Val := 0;
15     end Pop;
16
17 end Stacks;

```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Stacks
MD5: 364df7c6806af4a1bc957c2c2d53b2cc

In the above example, we define a stack type in the public part (known as the *visible part* of the package spec in Ada), but the exact representation of that type is private.

Then, in the private part, we define the representation of that type. We can also declare other types that will be used as *helpers* for our main public type. This is useful since declaring helper types is common in Ada.

A few words about terminology:

- The Stack type as viewed from the public part is called the partial view of the type. This is what clients have access to.
- The Stack type as viewed from the private part or the body of the package is called the full view of the type. This is what implementers have access to.

From the point of view of the client (the *with*'ing unit), only the public (visible) part is important, and the private part could as well not exist. It makes it very easy to read linearly the part of the package that is important for you.

```

-- No need to read the private part to use the package
package Stacks is
  type Stack is private;

  procedure Push (S : in out Stack;
                 Val : Integer);
  procedure Pop (S : in out Stack;
                Val : out Integer);
private
  ...
end Stacks;

```

Here is how the Stacks package would be used:

```

-- Example of use
with Stacks; use Stacks;

procedure Test_Stack is
  S : Stack;
  Res : Integer;
begin
  Push (S, 5);
  Push (S, 7);
  Pop (S, Res);
end Test_Stack;

```


12.3 Limited types

Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

Listing 144: stacks.ads

```
1 package Stacks is
2   type Stack is limited private;
3   -- Limited type. Cannot assign nor compare.
4
5   procedure Push (S : in out Stack;
6                  Val : Integer);
7   procedure Pop (S : in out Stack;
8                 Val : out Integer);
9 private
10  subtype Stack_Index is
11    Natural range 1 .. 10;
12
13  type Content_Type is
14    array (Stack_Index) of Natural;
15
16  type Stack is limited record
17    Top : Stack_Index;
18    Content : Content_Type;
19  end record;
20 end Stacks;
```

Listing 145: stacks.adb

```
1 package body Stacks is
2
3   procedure Push (S : in out Stack;
4                  Val : Integer) is
5   begin
6     -- Missing implementation!
7     null;
8   end Push;
9
10  procedure Pop (S : in out Stack;
11                Val : out Integer) is
12  begin
13    -- Dummy implementation!
14    Val := 0;
15  end Pop;
16
17 end Stacks;
```

Listing 146: main.adb

```

1 with Stacks; use Stacks;
2
3 procedure Main is
4   S, S2 : Stack;
5 begin
6   S := S2;
7   -- Illegal: S is limited.
8 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Limited_Stacks
MD5: 811343b46f20ac6af5e1bf26561f8d8d

Build output

```

main.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

```

This is useful because, for example, for some data types the built-in assignment operation might be incorrect (for example when a deep copy is required).

Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).

Ada also allows you to implement special semantics for assignment via [controlled types](#)¹⁶. However, in some cases assignment is simply inappropriate; one example is the **File_Type** from the Ada.Text_IO package, which is declared as a limited type and thus attempts to assign one file to another would be detected as illegal.

12.4 Child packages & privacy

We've seen previously (in the [child packages section](#) (page 39)) that packages can have child packages. Privacy plays an important role in child packages. This section discusses some of the privacy rules that apply to child packages.

Although the private part of a package P is meant to encapsulate information, certain parts of a child package P.C can have access to this private part of P. In those cases, information from the private part of P can then be used as if it were declared in the public part of its specification. To be more specific, the body of P.C and the private part of the specification of P.C have access to the private part of P. However, the public part of the specification of P.C only has access to the public part of P's specification. The following table summarizes this:

Part of a child package	Access to the private part of its parent's specification
Specification: public part	
Specification: private part	✓
Body	✓

The rest of this section shows examples of how this access to private information actually works for child packages.

Let's first look at an example where the body of a child package P.C has access to the private part of the specification of its parent P. We've seen, in a previous source-code example,

¹⁶ <http://www.ada-auth.org/standards/12rm/html/RM-7-6.html>

that the `Hello2` procedure declared in the private part of the `Encapsulate` package cannot be used in the `Main` procedure, since it's not visible there. This limitation doesn't apply, however, for parts of the child packages of the `Encapsulate` package. In fact, the body of its child package `Encapsulate.Child` has access to the `Hello2` procedure and can call it there, as you can see in the implementation of the `Hello3` procedure of the `Child` package:

Listing 147: `encapsulate.ads`

```
1 package Encapsulate is
2   procedure Hello;
3
4 private
5
6   procedure Hello2;
7   -- Not visible from external units
8   -- But visible in child packages
9 end Encapsulate;
```

Listing 148: `encapsulate.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate is
4
5   procedure Hello is
6   begin
7     Put_Line ("Hello");
8   end Hello;
9
10  procedure Hello2 is
11  begin
12    Put_Line ("Hello #2");
13  end Hello2;
14
15 end Encapsulate;
```

Listing 149: `encapsulate-child.ads`

```
1 package Encapsulate.Child is
2
3   procedure Hello3;
4
5 end Encapsulate.Child;
```

Listing 150: `encapsulate-child.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate.Child is
4
5   procedure Hello3 is
6   begin
7     -- Using private procedure Hello2
8     -- from the parent package
9     Hello2;
10    Put_Line ("Hello #3");
11  end Hello3;
12
13 end Encapsulate.Child;
```

Listing 151: main.adb

```

1 with Encapsulate.Child;
2
3 procedure Main is
4 begin
5     Encapsulate.Child.Hello3;
6 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Encapsulate_Child
MD5: 1533f43eee8f8b4d14c9b2101f42f13a

Runtime output

```

Hello #2
Hello #3

```

The same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package. Let's look at an example:

Listing 152: my_types.ads

```

1 package My_Types is
2
3     type Priv_Rec is private;
4
5 private
6
7     type Priv_Rec is record
8         Number : Integer := 42;
9     end record;
10
11 end My_Types;

```

Listing 153: my_types-ops.ads

```

1 package My_Types.Ops is
2
3     procedure Display (E : Priv_Rec);
4
5 end My_Types.Ops;

```

Listing 154: my_types-ops.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Types.Ops is
4
5     procedure Display (E : Priv_Rec) is
6     begin
7         Put_Line ("Priv_Rec.Number: "
8                 & Integer'Image (E.Number));
9     end Display;
10
11 end My_Types.Ops;

```

Listing 155: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with My_Types; use My_Types;
4 with My_Types.Ops; use My_Types.Ops;
5
6 procedure Main is
7   E : Priv_Rec;
8 begin
9   Put_Line ("Presenting information:");
10
11   -- The following code would trigger a
12   -- compilation error here:
13   --
14   -- Put_Line ("Priv_Rec.Number: "
15   --           & Integer'Image (E.Number));
16
17   Display (E);
18 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Privacy.Private_Type_Child
MD5: 9960611460bc1190b30949eca08fc02b

Runtime output

```
Presenting information:
Priv_Rec.Number: 42
```

In this example, we don't have access to the `Number` component of the record type `Priv_Rec` in the `Main` procedure. You can see this in the call to `Put_Line` that has been commented-out in the implementation of `Main`. Trying to access the `Number` component there would trigger a compilation error. But we do have access to this component in the body of the `My_Types.Ops` package, since it's a child package of the `My_Types` package. Therefore, `Ops`'s body has access to the declaration of the `Priv_Rec` type — which is in the private part of its parent, the `My_Types` package. For this reason, the same call to `Put_Line` that would trigger a compilation error in the `Main` procedure works fine in the `Display` procedure of the `My_Types.Ops` package.

This kind of privacy rules for child packages allows for extending the functionality of a parent package and, at the same time, retain its encapsulation.

As we mentioned previously, in addition to the package body, the private part of the specification of a child package `P.C` also has access to the private part of the specification of its parent `P`. Let's look at an example where we declare an object of private type `Priv_Rec` in the private part of the child package `My_Types.Child` and initialize the `Number` component of the `Priv_Rec` record directly:

```
package My_Types.Child is
private
  E : Priv_Rec := (Number => 99);
end My_Types.Ops;
```

As expected, we wouldn't be able to initialize this component if we moved this declaration to the public (visible) part of the same child package:

```
package My_Types.Child is  
    E : Priv_Rec := (Number => 99);  
end My_Types.Ops;
```

The declaration above triggers a compilation error, since type `Priv_Rec` is private. Because the public part of `My_Types.Child` is also visible outside the child package, Ada cannot allow accessing private information in this part of the specification.

GENERICICS

13.1 Introduction

Generics are used for metaprogramming in Ada. They are useful for abstract algorithms that share common properties with each other.

Either a subprogram or a package can be generic. A generic is declared by using the keyword **generic**. For example:

Listing 156: operator.ads

```
1 generic
2   type T is private;
3   -- Declaration of formal types and objects
4   -- Below, we could use one of the following:
5   -- <procedure | function | package>
6 procedure Operator (Dummy : in out T);
```

Listing 157: operator.adb

```
1 procedure Operator (Dummy : in out T) is
2 begin
3   null;
4 end Operator;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Simple_Generic
MD5: 1321d437043dafdb725fad416e654318

13.2 Formal type declaration

Formal types are abstractions of a specific type. For example, we may want to create an algorithm that works on any integer type, or even on any type at all, whether a numeric type or not. The following example declares a formal type T for the Set procedure.

Listing 158: set.ads

```
1 generic
2   type T is private;
3   -- T is a formal type that indicates that
4   -- any type can be used, possibly a numeric
5   -- type or possibly even a record type.
6 procedure Set (Dummy : T);
```


Listing 159: set.adb

```
1 procedure Set (Dummy : T) is
2 begin
3     null;
4 end Set;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Forma_l_Type_Declaration
MD5: 668156f66b2479c4932d18b5ad35deba

The declaration of T as **private** indicates that you can map any definite type to it. But you can also restrict the declaration to allow only some types to be mapped to that formal type. Here are some examples:

Formal Type	Format
Any type	type T is private;
Any discrete type	type T is (<>);
Any floating-point type	type T is digits <>;

13.3 Formal object declaration

Formal objects are similar to subprogram parameters. They can reference formal types declared in the formal specification. For example:

Listing 160: set.ads

```
1 generic
2     type T is private;
3     X : in out T;
4     -- X can be used in the Set procedure
5 procedure Set (E : T);
```

Listing 161: set.adb

```
1 procedure Set (E : T) is
2     pragma Unreferenced (E, X);
3 begin
4     null;
5 end Set;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Forma_l_Object_Declaration
MD5: 1b88bc0e5b8f48a35394966e6af07ac0

Formal objects can be either input parameters or specified using the **in out** mode.

13.4 Generic body definition

We don't repeat the **generic** keyword for the body declaration of a generic subprogram or package. Instead, we start with the actual declaration and use the generic types and objects we declared. For example:

Listing 162: set.ads

```

1 generic
2   type T is private;
3   X : in out T;
4 procedure Set (E : T);

```

Listing 163: set.adb

```

1 procedure Set (E : T) is
2   -- Body definition: "generic" keyword
3   -- is not used
4 begin
5   X := E;
6 end Set;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Generic_Body_Definition
MD5: de611ef77b528543fd6bad82c53857f7

13.5 Generic instantiation

Generic subprograms or packages can't be used directly. Instead, they need to be instantiated, which we do using the **new** keyword, as shown in the following example:

Listing 164: set.ads

```

1 generic
2   type T is private;
3   X : in out T;
4   -- X can be used in the Set procedure
5 procedure Set (E : T);

```

Listing 165: set.adb

```

1 procedure Set (E : T) is
2 begin
3   X := E;
4 end Set;

```

Listing 166: show_generic_instantiation.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Set;
3
4 procedure Show_Generic_Instantiation is
5
6   Main      : Integer := 0;
7   Current  : Integer;
8
9   procedure Set_Main is new Set (T => Integer,
10                                X => Main);
11   -- Here, we map the formal parameters to
12   -- actual types and objects.
13   --
14   -- The same approach can be used to
15   -- instantiate functions or packages, e.g.:

```

(continues on next page)

(continued from previous page)

```

16  --
17  --  function Get_Main is new ...
18  --  package Integer_Queue is new ...
19
20  begin
21      Current := 10;
22
23      Set_Main (Current);
24      Put_Line ("Value of Main is "
25              & Integer'Image (Main));
26  end Show_Generic_Instantiation;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Generic_Instantiation
 MD5: 13dc0692252496d954240952561e1c05

Runtime output

Value of Main is 10

In the example above, we instantiate the procedure Set by mapping the formal parameters T and X to actual existing elements, in this case the **Integer** type and the Main variable.

13.6 Generic packages

The previous examples focused on generic subprograms. In this section, we look at generic packages. The syntax is similar to that used for generic subprograms: we start with the **generic** keyword and continue with formal declarations. The only difference is that **package** is specified instead of a subprogram keyword.

Here's an example:

Listing 167: element.ads

```

1  generic
2      type T is private;
3  package Element is
4
5      procedure Set (E : T);
6      procedure Reset;
7      function Get return T;
8      function Is_Valid return Boolean;
9
10     Invalid_Element : exception;
11
12 private
13     Value : T;
14     Valid : Boolean := False;
15 end Element;

```

Listing 168: element.adb

```

1  package body Element is
2
3      procedure Set (E : T) is
4      begin
5          Value := E;
6          Valid := True;

```

(continues on next page)

(continued from previous page)

```

7   end Set;
8
9   procedure Reset is
10  begin
11      Valid := False;
12  end Reset;
13
14  function Get return T is
15  begin
16      if not Valid then
17          raise Invalid_Element;
18      end if;
19      return Value;
20  end Get;
21
22  function Is_Valid return Boolean is (Valid);
23 end Element;

```

Listing 169: show_generic_package.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Element;
3
4  procedure Show_Generic_Package is
5
6      package I is new Element (T => Integer);
7
8      procedure Display_Initialized is
9      begin
10         if I.Is_Valid then
11             Put_Line ("Value is initialized");
12         else
13             Put_Line ("Value is not initialized");
14         end if;
15     end Display_Initialized;
16
17 begin
18     Display_Initialized;
19
20     Put_Line ("Initializing...");
21     I.Set (5);
22     Display_Initialized;
23     Put_Line ("Value is now set to "
24              & Integer'Image (I.Get));
25
26     Put_Line ("Resetting...");
27     I.Reset;
28     Display_Initialized;
29
30 end Show_Generic_Package;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Generic_Package
MD5: c5278a06c6d06f1f37353ee0ca6686ec

Runtime output

```

Value is not initialized
Initializing...
Value is initialized

```

(continues on next page)

(continued from previous page)

```
Value is now set to 5
Resetting...
Value is not initialized
```

In the example above, we created a simple container named `Element`, with just one single element. This container tracks whether the element has been initialized or not.

After writing the package definition, we create the instance `I` of the `Element`. We use the instance by calling the package subprograms (`Set`, `Reset`, and `Get`).

13.7 Formal subprograms

In addition to formal types and objects, we can also declare formal subprograms or packages. This course only describes formal subprograms; formal packages are discussed in the advanced course.

We use the `with` keyword to declare a formal subprogram. In the example below, we declare a formal function (`Comparison`) to be used by the generic procedure `Check`.

Listing 170: check.ads

```
1 generic
2   Description : String;
3   type T is private;
4   with function Comparison (X, Y : T)
5     return Boolean;
6 procedure Check (X, Y : T);
```

Listing 171: check.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Check (X, Y : T) is
4   Result : Boolean;
5 begin
6   Result := Comparison (X, Y);
7   if Result then
8     Put_Line
9       ("Comparison ("
10        & Description
11        & ") between arguments is OK!");
12   else
13     Put_Line
14       ("Comparison ("
15        & Description
16        & ") between arguments is not OK!");
17   end if;
18 end Check;
```

Listing 172: show_formal_subprogram.adb

```
1 with Check;
2
3 procedure Show_Formalsubprogram is
4
5   A, B : Integer;
6
7   procedure Check_Is_Equal is new
8     Check (Description => "equality",
```

(continues on next page)

(continued from previous page)

```

9         T           => Integer,
10        Comparison => Standard."=");
11    -- Here, we are mapping the standard
12    -- equality operator for Integer types to
13    -- the Comparison formal function
14    begin
15        A := 0;
16        B := 1;
17        Check_Is_Equal (A, B);
18    end Show_Forma_Subprogram;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Forma_Subprogram
MD5: 1c463a47e9ce56b5afbca1da6acd116d

Runtime output

Comparison (equality) between arguments is not OK!

13.8 Example: I/O instances

Ada offers generic I/O packages that can be instantiated for standard and derived types. One example is the generic Float_IO package, which provides procedures such as Put and Get. In fact, Float_Text_IO — available from the standard library — is an instance of the Float_IO package, and it's defined as:

```

with Ada.Text_IO;

package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO (Float);

```

You can use it directly with any object of floating-point type. For example:

Listing 173: show_float_text_io.adb

```

1  with Ada.Float_Text_IO;
2
3  procedure Show_Float_Text_IO is
4      X : constant Float := 2.5;
5
6      use Ada.Float_Text_IO;
7  begin
8      Put (X);
9  end Show_Float_Text_IO;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Float_Text_IO
MD5: 7cc9b547ef301a2071e9fb65caa4631b

Runtime output

2.50000E+00

Instantiating generic I/O packages can be useful for derived types. For example, let's create a new type Price that must be displayed with two decimal digits after the point, and no exponent.

Listing 174: show_float_io_inst.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Float_IO_Inst is
4
5     type Price is digits 3;
6
7     package Price_IO is new
8         Ada.Text_IO.Float_IO (Price);
9
10    P : Price;
11 begin
12     -- Set to zero => don't display exponent
13     Price_IO.Default_Exp := 0;
14
15     P := 2.5;
16     Price_IO.Put (P);
17     New_Line;
18
19     P := 5.75;
20     Price_IO.Put (P);
21     New_Line;
22 end Show_Float_IO_Inst;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Float_IO_Inst
MD5: 583c761421d7fdb812dd2a183b676bae

Runtime output

```
2.50
5.75
```

By adjusting `Default_Exp` from the `Price_IO` instance to *remove* the exponent, we can control how variables of `Price` type are displayed. Just as a side note, we could also have written:

```
-- [...]
```

```
type Price is new Float;

package Price_IO is new
    Ada.Text_IO.Float_IO (Price);

begin
    Price_IO.Default_Aft := 2;
    Price_IO.Default_Exp := 0;
```

In this case, we're adjusting `Default_Aft`, too, to get two decimal digits after the point when calling `Put`.

In addition to the generic `Float_IO` package, the following generic packages are available from `Ada.Text_IO`:

- `Enumeration_IO` for enumeration types;
- `Integer_IO` for integer types;
- `Modular_IO` for modular types;
- `Fixed_IO` for fixed-point types;

- Decimal_IO for decimal types.

In fact, we could rewrite the example above using decimal types:

Listing 175: show_decimal_io_inst.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_IO_Inst is
4
5     type Price is delta 10.0 ** (-2) digits 12;
6
7     package Price_IO is new
8         Ada.Text_IO.Decimal_IO (Price);
9
10    P : Price;
11 begin
12    Price_IO.Default_Exp := 0;
13
14    P := 2.5;
15    Price_IO.Put (P);
16    New_Line;
17
18    P := 5.75;
19    Price_IO.Put (P);
20    New_Line;
21 end Show_Decimal_IO_Inst;
```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Decimal_IO_Inst
MD5: f413570759dcb32cc166078b3ceel1a16

Runtime output

```

2.50
5.75
```

13.9 Example: ADTs

An important application of generics is to model abstract data types (ADTs). In fact, Ada includes a library with numerous ADTs using generics: `Ada.Containers` (described in the [containers section](#) (page 205)).

A typical example of an ADT is a stack:

Listing 176: stacks.ads

```

1 generic
2     Max : Positive;
3     type T is private;
4 package Stacks is
5
6     type Stack is limited private;
7
8     Stack_Underflow, Stack_Overflow : exception;
9
10    function Is_Empty (S : Stack) return Boolean;
11
12    function Pop (S : in out Stack) return T;
13
```

(continues on next page)

(continued from previous page)

```

14  procedure Push (S : in out Stack;
15                V :          T);
16
17  private
18
19  type Stack_Array is
20    array (Natural range <>) of T;
21
22  Min : constant := 1;
23
24  type Stack is record
25    Container : Stack_Array (Min .. Max);
26    Top       : Natural := Min - 1;
27  end record;
28
29  end Stacks;

```

Listing 177: stacks.adb

```

1  package body Stacks is
2
3    function Is_Empty (S : Stack) return Boolean is
4      (S.Top < S.Container'First);
5
6    function Is_Full (S : Stack) return Boolean is
7      (S.Top >= S.Container'Last);
8
9    function Pop (S : in out Stack) return T is
10   begin
11     if Is_Empty (S) then
12       raise Stack_Underflow;
13     else
14       return X : T do
15         X := S.Container (S.Top);
16         S.Top := S.Top - 1;
17       end return;
18     end if;
19   end Pop;
20
21   procedure Push (S : in out Stack;
22                 V :          T) is
23   begin
24     if Is_Full (S) then
25       raise Stack_Overflow;
26     else
27       S.Top := S.Top + 1;
28       S.Container (S.Top) := V;
29     end if;
30   end Push;
31
32 end Stacks;

```

Listing 178: show_stack.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Stacks;
3
4  procedure Show_Stack is
5
6    package Integer_Stacks is new

```

(continues on next page)

(continued from previous page)

```

7      Stacks (Max => 10,
8             T   => Integer);
9      use Integer_Stacks;
10
11     Values : Integer_Stacks.Stack;
12
13 begin
14     Push (Values, 10);
15     Push (Values, 20);
16
17     Put_Line ("Last value was "
18             & Integer'Image (Pop (Values)));
19 end Show_Stack;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Show_Stack
MD5: ee112d395552c1a02d211b9e5425dc71

Runtime output

Last value was 20

In this example, we first create a generic stack package (Stacks) and then instantiate it to create a stack of up to 10 integer values.

13.10 Example: Swap

Let's look at a simple procedure that swaps variables of type Color:

Listing 179: colors.ads

```

1 package Colors is
2     type Color is (Black, Red, Green,
3                   Blue, White);
4
5     procedure Swap_Colors (X, Y : in out Color);
6 end Colors;

```

Listing 180: colors.adb

```

1 package body Colors is
2
3     procedure Swap_Colors (X, Y : in out Color) is
4         Tmp : constant Color := X;
5     begin
6         X := Y;
7         Y := Tmp;
8     end Swap_Colors;
9
10 end Colors;

```

Listing 181: test_non_generic_swap_colors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Test_Non_Generic_Swap_Colors is

```

(continues on next page)

(continued from previous page)

```

5   A, B, C : Color;
6   begin
7   A := Blue;
8   B := White;
9   C := Red;
10
11   Put_Line ("Value of A is "
12             & Color'Image (A));
13   Put_Line ("Value of B is "
14             & Color'Image (B));
15   Put_Line ("Value of C is "
16             & Color'Image (C));
17
18   New_Line;
19   Put_Line ("Swapping A and C...");
20   New_Line;
21   Swap_Colors (A, C);
22
23   Put_Line ("Value of A is "
24             & Color'Image (A));
25   Put_Line ("Value of B is "
26             & Color'Image (B));
27   Put_Line ("Value of C is "
28             & Color'Image (C));
29   end Test_Non_Generic_Swap_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Non_Generic_Swap_Colors
MD5: 4d1cf826a1676c3750a8aabd484ac71f

Runtime output

```

Value of A is BLUE
Value of B is WHITE
Value of C is RED

Swapping A and C...

Value of A is RED
Value of B is WHITE
Value of C is BLUE

```

In this example, `Swap_Colors` can only be used for the `Color` type. However, this algorithm can theoretically be used for any type, whether an enumeration type or a complex record type with many elements. The algorithm itself is the same: it's only the type that differs. If, for example, we want to swap variables of **Integer** type, we don't want to duplicate the implementation. Therefore, such an algorithm is a perfect candidate for abstraction using generics.

In the example below, we create a generic version of `Swap_Colors` and name it **Generic_Swap**. This generic version can operate on any type due to the declaration of formal type `T`.

Listing 182: generic_swap.ads

```

1   generic
2     type T is private;
3   procedure Generic_Swap (X, Y : in out T);

```

Listing 183: generic_swap.adb

```

1  procedure Generic_Swap (X, Y : in out T) is
2      Tmp : constant T := X;
3  begin
4      X := Y;
5      Y := Tmp;
6  end Generic_Swap;

```

Listing 184: colors.ads

```

1  with Generic_Swap;
2
3  package Colors is
4
5      type Color is (Black, Red, Green,
6                    Blue, White);
7
8      procedure Swap_Colors is new
9          Generic_Swap (T => Color);
10
11 end Colors;

```

Listing 185: test_swap_colors.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Colors; use Colors;
3
4  procedure Test_Swap_Colors is
5      A, B, C : Color;
6  begin
7      A := Blue;
8      B := White;
9      C := Red;
10
11      Put_Line ("Value of A is "
12              & Color'Image (A));
13      Put_Line ("Value of B is "
14              & Color'Image (B));
15      Put_Line ("Value of C is "
16              & Color'Image (C));
17
18      New_Line;
19      Put_Line ("Swapping A and C...");
20      New_Line;
21      Swap_Colors (A, C);
22
23      Put_Line ("Value of A is "
24              & Color'Image (A));
25      Put_Line ("Value of B is "
26              & Color'Image (B));
27      Put_Line ("Value of C is "
28              & Color'Image (C));
29  end Test_Swap_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Swap_Colors
MD5: a5d94a40bd9d1c6736cc873f8b58e867

Runtime output

```
Value of A is BLUE
Value of B is WHITE
Value of C is RED
```

```
Swapping A and C...
```

```
Value of A is RED
Value of B is WHITE
Value of C is BLUE
```

As we can see in the example, we can create the same `Swap_Colors` procedure as we had in the non-generic version of the algorithm by declaring it as an instance of the generic `Generic_Swap` procedure. We specify that the generic `T` type will be mapped to the `Color` type by passing it as an argument to the `Generic_Swap` instantiation.

13.11 Example: Reversing

The previous example, with an algorithm to swap two values, is one of the simplest examples of using generics. Next we study an algorithm for reversing elements of an array. First, let's start with a non-generic version of the algorithm, one that works specifically for the `Color` type:

Listing 186: colors.ads

```
1 package Colors is
2
3     type Color is (Black, Red, Green,
4                   Blue, White);
5
6     type Color_Array is
7       array (Integer range <>) of Color;
8
9     procedure Reverse_It (X : in out Color_Array);
10
11 end Colors;
```

Listing 187: colors.adb

```
1 package body Colors is
2
3     procedure Reverse_It (X : in out Color_Array)
4     is
5     begin
6         for I in X'First ..
7             (X'Last + X'First) / 2 loop
8             declare
9                 Tmp      : Color;
10                X_Left   : Color
11                renames X (I);
12                X_Right  : Color
13                renames X (X'Last + X'First - I);
14            begin
15                Tmp      := X_Left;
16                X_Left   := X_Right;
17                X_Right  := Tmp;
18            end;
19        end loop;
20    end Reverse_It;
21
22 end Colors;
```

Listing 188: test_non_generic_reverse_colors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Test_Non_Generic_Reverse_Colors is
5
6     My_Colors : Color_Array (1 .. 5) :=
7         (Black, Red, Green, Blue, White);
8
9 begin
10     for C of My_Colors loop
11         Put_Line ("My_Color: " & Color'Image (C));
12     end loop;
13
14     New_Line;
15     Put_Line ("Reversing My_Color...");
16     New_Line;
17     Reverse_It (My_Colors);
18
19     for C of My_Colors loop
20         Put_Line ("My_Color: " & Color'Image (C));
21     end loop;
22
23 end Test_Non_Generic_Reverse_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Non_Generic_Reverse_Colors
MD5: 9b3a489d0bc0ecd79de6ba99fd7cd44f

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

```

The procedure `Reverse_It` takes an array of colors, starts by swapping the first and last elements of the array, and continues doing that with successive elements until it reaches the middle of array. At that point, the entire array has been reversed, as we see from the output of the test program.

To abstract this procedure, we declare formal types for three components of the algorithm:

- the elements of the array (Color type in the example)
- the range used for the array (**Integer** range in the example)
- the actual array type (Color_Array type in the example)

This is a generic version of the algorithm:

Listing 189: generic_reverse.ads

```
1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6   procedure Generic_Reverse (X : in out Array_T);
```

Listing 190: generic_reverse.adb

```
1 procedure Generic_Reverse (X : in out Array_T) is
2 begin
3   for I in X'First ..
4     (X'Last + X'First) / 2 loop
5     declare
6       Tmp      : T;
7       X_Left   : T;
8       renames X (I);
9       X_Right  : T;
10      renames X (X'Last + X'First - I);
11    begin
12      Tmp      := X_Left;
13      X_Left   := X_Right;
14      X_Right  := Tmp;
15    end;
16  end loop;
17 end Generic_Reverse;
```

Listing 191: colors.ads

```
1 with Generic_Reverse;
2
3 package Colors is
4
5   type Color is (Black, Red, Green,
6                 Blue, White);
7
8   type Color_Array is
9     array (Integer range <>) of Color;
10
11   procedure Reverse_It is new
12     Generic_Reverse (T      => Color,
13                     Index   => Integer,
14                     Array_T => Color_Array);
15
16 end Colors;
```

Listing 192: test_reverse_colors.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Test_Reverse_Colors is
5
6   My_Colors : Color_Array (1 .. 5) :=
7     (Black, Red, Green, Blue, White);
8
9 begin
10   for C of My_Colors loop
```

(continues on next page)

(continued from previous page)

```

11     Put_Line ("My_Color: "
12               & Color'Image (C));
13 end loop;
14
15 New_Line;
16 Put_Line ("Reversing My_Color...");
17 New_Line;
18 Reverse_It (My_Colors);
19
20 for C of My_Colors loop
21     Put_Line ("My_Color: "
22               & Color'Image (C));
23 end loop;
24
25 end Test_Reverse_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Reverse_Colors
MD5: 9ef175c517d7574b4b65b24ba0027f1f

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

```

As mentioned above, we're abstracting three components of the algorithm:

- the T type abstracts the elements of the array
- the Index type abstracts the range used for the array
- the Array_T type abstracts the array type and uses the formal declarations of the T and Index types.

13.12 Example: Test application

In the previous example we've focused only on abstracting the reversing algorithm itself. However, we could have decided to also abstract our small test application. This could be useful if we, for example, decide to test other procedures that change elements of an array.

In order to do this, we again have to choose the elements to abstract. We therefore declare the following formal parameters:

- S: the string containing the array name
- a function Image that converts an element of type T to a string
- a procedure Test that performs some operation on the array

Note that Image and Test are examples of formal subprograms and S is an example of a formal object.

Here is a version of the test application making use of the generic Perform_Test procedure:

Listing 193: generic_reverse.ads

```
1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6   procedure Generic_Reverse (X : in out Array_T);
```

Listing 194: generic_reverse.adb

```
1 procedure Generic_Reverse (X : in out Array_T) is
2 begin
3   for I in X'First ..
4     (X'Last + X'First) / 2 loop
5     declare
6       Tmp      : T;
7       X_Left   : T;
8       renames X (I);
9       X_Right  : T;
10      renames X (X'Last + X'First - I);
11    begin
12      Tmp      := X_Left;
13      X_Left   := X_Right;
14      X_Right  := Tmp;
15    end;
16  end loop;
17 end Generic_Reverse;
```

Listing 195: perform_test.ads

```
1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6   S : String;
7   with function Image (E : T)
8     return String is <>;
9   with procedure Test (X : in out Array_T);
10  procedure Perform_Test (X : in out Array_T);
```

Listing 196: perform_test.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Perform_Test (X : in out Array_T) is
4 begin
5   for C of X loop
6     Put_Line (S & ": " & Image (C));
7   end loop;
8
9   New_Line;
10  Put_Line ("Testing " & S & "...");
11  New_Line;
12  Test (X);
13
14  for C of X loop
15    Put_Line (S & ": " & Image (C));
```

(continues on next page)

(continued from previous page)

```

16   end loop;
17 end Perform_Test;

```

Listing 197: colors.ads

```

1  with Generic_Reverse;
2
3  package Colors is
4
5      type Color is (Black, Red, Green,
6                    Blue, White);
7
8      type Color_Array is
9          array (Integer range <>) of Color;
10
11     procedure Reverse_It is new
12         Generic_Reverse (T      => Color,
13                         Index   => Integer,
14                         Array_T => Color_Array);
15
16 end Colors;

```

Listing 198: test_reverse_colors.adb

```

1  with Colors;      use Colors;
2  with Perform_Test;
3
4  procedure Test_Reverse_Colors is
5
6      procedure Perform_Test_Reverse_It is new
7          Perform_Test (T      => Color,
8                      Index   => Integer,
9                      Array_T => Color_Array,
10                     S      => "My_Color",
11                     Image   => Color'Image,
12                     Test    => Reverse_It);
13
14     My_Colors : Color_Array (1 .. 5) :=
15         (Black, Red, Green, Blue, White);
16
17 begin
18     Perform_Test_Reverse_It (My_Colors);
19 end Test_Reverse_Colors;

```

Code block metadata

Project: Courses.Intro_To_Ada.Generics.Test_Reverse_Colors_2
MD5: 04640309f4f7e9f8bcff137d1a6f8733

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Testing My_Color...

My_Color: WHITE

```

(continues on next page)

(continued from previous page)

```
My_Color: BLUE  
My_Color: GREEN  
My_Color: RED  
My_Color: BLACK
```

In this example, we create the procedure `Perform_Test_Reverse_It` as an instance of the generic procedure (`Perform_Test`). Note that:

- For the formal `Image` function, we use the '`Image`' attribute of the `Color` type
- For the formal `Test` procedure, we reference the `Reverse_Array` procedure from the package.

EXCEPTIONS

Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about *raising*, not *throwing*, an exception and *handling*, not *catching*, an exception.

14.1 Exception declaration

Ada exceptions are not types, but instead objects, which may be peculiar to you if you're used to the way Java or Python support exceptions. Here's how you declare an exception:

Listing 199: exceptions.ads

```
1 package Exceptions is
2     My_Except : exception;
3     -- Like an object. *NOT* a type !
4 end Exceptions;
```

Code block metadata

Project: Courses.Intro_To_Ada.Exceptions.Show_Exception
MD5: 6201faeca9b029c790023856d2c8c419

Even though they're objects, you're going to use each declared exception object as a "kind" or "family" of exceptions. Ada does not require that a subprogram declare every exception it can potentially raise.

14.2 Raising an exception

To raise an exception of our newly declared exception kind, do the following:

Listing 200: main.adb

```
1 with Exceptions; use Exceptions;
2
3 procedure Main is
4 begin
5     raise My_Except;
6     -- Execution of current control flow
7     -- abandoned; an exception of kind
8     -- "My_Except" will bubble up until it
9     -- is caught.
10 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Exceptions.Show_Exception
MD5: 24b40ae1509722adf51c3dd0d3ea4fbe

Runtime output

```
raised EXCEPTIONS.MY_EXCEPT : main.adb:5
```

Here, the `My_Except` exception is raised. We can also specify a message:

Listing 201: main.adb

```
1 with Exceptions; use Exceptions;
2
3 procedure Main is
4 begin
5     raise My_Except with "My exception message";
6     -- Execution of current control flow
7     -- abandoned; an exception of kind
8     -- "My_Except" with associated string will
9     -- bubble up until it is caught.
10 end Main;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Exceptions.Show_Exception
MD5: 279299c9703c3ed4e51fdd7c3a5e1392
```

Runtime output

```
raised EXCEPTIONS.MY_EXCEPT : My exception message
```

In this case, we see an additional message when the exception is displayed.

14.3 Handling an exception

Next, we address how to handle exceptions that were raised by us or libraries that we call. The neat thing in Ada is that you can add an exception handler to any statement block as follows:

Listing 202: open_file.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Open_File is
5     File : File_Type;
6 begin
7     -- Block (sequence of statements)
8     begin
9         Open (File, In_File, "input.txt");
10    exception
11        when E : Name_Error =>
12            -- ^ Exception to be handled
13            Put ("Cannot open input file : ");
14            Put_Line (Exception_Message (E));
15            raise;
16            -- Reraise current occurrence
17    end;
18 end Open_File;
```

Code block metadata

Project: Courses.Intro_To_Ada.Exceptions.Show_Exception_Handling
 MD5: 4eald5da684a6d7d7ee32908810e9c8f

Runtime output

```
Cannot open input file : input.txt: No such file or directory
raised ADA.IO_EXCEPTIONS.NAME_ERROR : input.txt: No such file or directory
```

In the example above, we're using the `Exception_Message` function from the `Ada.Exceptions` package. This function returns the message associated with the exception as a string.

You don't need to introduce a block just to handle an exception: you can add it to the statements block of your current subprogram:

Listing 203: `open_file.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Open_File is
5   File : File_Type;
6 begin
7   Open (File, In_File, "input.txt");
8   -- Exception block can be added to any block
9 exception
10  when Name_Error =>
11    Put ("Cannot open input file");
12 end Open_File;
```

Code block metadata

Project: Courses.Intro_To_Ada.Exceptions.Show_Exception_Message
 MD5: 838e87ae416b3a717901cdc00eb71b40

Runtime output

```
Cannot open input file
```

Attention

Exception handlers have an important restriction that you need to be careful about: Exceptions raised in the declarative section are not caught by the handlers of that block. So for example, in the following code, the exception will not be caught.

Listing 204: `be_careful.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Be_Careful is
5   function Dangerous return Integer is
6   begin
7     raise Constraint_Error;
8     return 42;
9   end Dangerous;
10
11 begin
12   declare
13     A : Integer := Dangerous;
```

```
14 begin
15     Put_Line (Integer'Image (A));
16 exception
17     when Constraint_Error =>
18         Put_Line ("error!");
19 end;
20 end Be_Careful;
```

Code block metadata

Project: Courses.Intro_To_Ada.Exceptions.Be_Careful
MD5: 6ea8a214bbbaca09d7444136d069e782

Runtime output

raised CONSTRAINT_ERROR : be_careful.adb:7 explicit raise

This is also the case for the top-level exception block that is part of the current subprogram.

14.4 Predefined exceptions

Ada has a very small number of predefined exceptions:

- `Constraint_Error` is the main one you might see. It's raised:
 - When bounds don't match or, in general, any violation of constraints.
 - In case of overflow
 - In case of null dereferences
 - In case of division by 0
- `Program_Error` might appear, but probably less often. It's raised in more arcane situations, such as for order of elaboration issues and some cases of detectable erroneous execution.
- `Storage_Error` will happen because of memory issues, such as:
 - Not enough memory (allocator)
 - Not enough stack
- **Tasking_Error** will happen with task related errors, such as any error happening during task activation.

You should not reuse predefined exceptions. If you do then, it won't be obvious when one is raised that it is because something went wrong in a built-in language operation.

TASKING

Tasks and protected objects allow the implementation of concurrency in Ada. The following sections explain these concepts in more detail.

15.1 Tasks

A task can be thought as an application that runs *concurrently* with the main application. In other programming languages, a task might be called a [thread](#)¹⁷, and tasking might be called [multithreading](#)¹⁸.

Tasks may synchronize with the main application but may also process information completely independently from the main application. Here we show how this is accomplished.

15.1.1 Simple task

Tasks are declared using the keyword **task**. The task implementation is specified in a **task body** block. For example:

Listing 205: show_simple_task.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task is
4   task T;
5
6   task body T is
7     begin
8       Put_Line ("In task T");
9     end T;
10  begin
11    Put_Line ("In main");
12  end Show_Simple_Task;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Simple_Task
MD5: b17d9b35b4b2b53bc59776749e1be219

Runtime output

```
In task T
In main
```

¹⁷ [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))

¹⁸ [https://en.wikipedia.org/wiki/Thread_\(computing\)#Multithreading](https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading)

Here, we're declaring and implementing the task T. As soon as the main application starts, task T starts automatically — it's not necessary to manually start this task. By running the application above, we can see that both calls to `Put_Line` are performed.

Note that:

- The main application is itself a task (the main or “environment” task).
 - In this example, the subprogram `Show_Simple_Task` is the main task of the application.
- Task T is a subtask.
 - Each subtask has a master, which represents the program construct in which the subtask is declared. In this case, the main subprogram `Show_Simple_Task` is T's master.
 - The master construct is executed by some enclosing task, which we will refer to as the “master task” of the subtask.
- The number of tasks is not limited to one: we could include a task T2 in the example above.
 - This task also starts automatically and runs *concurrently* with both task T and the main task. For example:

Listing 206: `show_simple_tasks.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Tasks is
4   task T;
5   task T2;
6
7   task body T is
8   begin
9     Put_Line ("In task T");
10  end T;
11
12  task body T2 is
13  begin
14    Put_Line ("In task T2");
15  end T2;
16
17 begin
18   Put_Line ("In main");
19 end Show_Simple_Tasks;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Multiple_Simple_Task
MD5: 5e24b797e742bec306ad498f4f40d2b4

Runtime output

```
In task T
In task T2
In main
```

15.1.2 Simple synchronization

As we've just seen, as soon as the master construct reaches its “begin”, its subtasks also start automatically. The master continues its processing until it has nothing more to do. At that point, however, it will not terminate. Instead, the master waits until its subtasks have

finished before it allows itself to complete. In other words, this waiting process provides synchronization between the master task and its subtasks. After this synchronization, the master construct will complete. For example:

Listing 207: show_simple_sync.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Sync is
4     task T;
5     task body T is
6     begin
7         for I in 1 .. 10 loop
8             Put_Line ("hello");
9         end loop;
10    end T;
11 begin
12     null;
13     -- Will wait here until all tasks
14     -- have terminated
15 end Show_Simple_Sync;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Simple_Sync
MD5: 84afce465854f99f8cbe0b57714d8a5f

Runtime output

```

hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

The same mechanism is used for other subprograms that contain subtasks: the subprogram execution will wait for its subtasks to finish. So this mechanism is not limited to the main subprogram and also applies to any subprogram called by the main subprogram, directly or indirectly.

Synchronization also occurs if we move the task to a separate package. In the example below, we declare a task T in the package Simple_Sync_Pkg.

Listing 208: simple_sync_pkg.ads

```

1 package Simple_Sync_Pkg is
2     task T;
3 end Simple_Sync_Pkg;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: 2f9be044d04994240970f150e2293d5e

This is the corresponding package body:

Listing 209: simple_sync_pkg.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Sync_Pkg is
4     task body T is
5     begin
6         for I in 1 .. 10 loop
7             Put_Line ("hello");
8         end loop;
9     end T;
10 end Simple_Sync_Pkg;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: b668451e4fb10e802f619889bcd743ff

Because the package is **with**'ed by the main procedure, the task T defined in the package will become a subtask of the main task. For example:

Listing 210: test_simple_sync_pkg.adb

```
1 with Simple_Sync_Pkg;
2
3 procedure Test_Simple_Sync_Pkg is
4 begin
5     null;
6     -- Will wait here until all tasks
7     -- have terminated
8 end Test_Simple_Sync_Pkg;
```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Simple_Sync_Pkg
MD5: e51565b91767ce198496ef3e9c582ac8

Runtime output

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

As soon as the main subprogram returns, the main task synchronizes with any subtasks spawned by packages T from Simple_Sync_Pkg before finally terminating.

15.1.3 Delay

We can introduce a delay by using the keyword **delay**. This puts the current task to sleep for the length of time (in seconds) specified in the delay statement. For example:

Listing 211: show_delay.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Delay is
4
5     task T;
6
7     task body T is
8     begin
9         for I in 1 .. 5 loop
10             Put_Line ("hello from task T");
11             delay 1.0;
12             -- ^ Wait 1.0 seconds
13         end loop;
14     end T;
15 begin
16     delay 1.5;
17     Put_Line ("hello from main");
18 end Show_Delay;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Delay
MD5: 4a6e8039744301a128e8fb2dd27902a5

Runtime output

```

hello from task T
hello from task T
hello from main
hello from task T
hello from task T
hello from task T

```

In this example, we're making the task T wait one second after each time it displays the "hello" message. In addition, the main task is waiting 1.5 seconds before displaying its own "hello" message

15.1.4 Synchronization: rendezvous

The only type of synchronization we've seen so far is the one that happens automatically at the end of a master construct with a subtask. You can also define custom synchronization points using the keyword **entry**. An *entry* can be viewed as a special kind of subprogram, which is called by another task using a similar syntax, as we will see later.

In the task body definition, you define which part of the task will accept the entries by using the keyword **accept**. A task proceeds until it reaches an **accept** statement and then waits for some other task to synchronize with it. Specifically,

- The task with the entry waits at that point (in the **accept** statement), ready to accept a call to the corresponding entry from the master task.
- The other task calls the task entry, in a manner similar to a procedure call, to synchronize with the entry.

This synchronization between tasks is called a *rendezvous*. Let's see an example:

Listing 212: show_rendezvous.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Rendezvous is
4
5      task T is
6          entry Start;
7      end T;
8
9      task body T is
10         begin
11             accept Start;
12             --      ^ Waiting for somebody
13             --      to call the entry
14
15             Put_Line ("In T");
16         end T;
17
18     begin
19         Put_Line ("In Main");
20
21         -- Calling T's entry:
22         T.Start;
23     end Show_Rendezvous;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Rendezvous
MD5: 479eea7adc876ac359ad20ac6e3acf66

Runtime output

```

In Main
In T

```

In this example, we declare an entry `Start` for task `T`. In the task body, we implement this entry using `accept Start`. When task `T` reaches this point, it waits for some other task to call its entry. This synchronization occurs in the `T.Start` statement. After the rendezvous completes, the main task and task `T` again run concurrently until they synchronize one final time when the main subprogram `Show_Rendezvous` finishes.

An entry may be used to perform more than a simple task synchronization: it also may perform multiple statements during the time both tasks are synchronized. We do this with a `do ... end` block. For the previous example, we would simply write `accept Start do <statements>; end`. We use this kind of block in the next example.

15.1.5 Select loop

There's no limit to the number of times an entry can be accepted. We could even create an infinite loop in the task and accept calls to the same entry over and over again. An infinite loop, however, prevents the subtask from finishing, so it blocks its master task when it reaches the end of its processing. Therefore, a loop containing `accept` statements in a task body can be used in conjunction with a `select ... or terminate` statement. In simple terms, this statement allows its master task to automatically terminate the subtask when the master construct reaches its end. For example:

Listing 213: show_rendezvous_loop.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```

3  procedure Show_Rendezvous_Loop is
4
5      task T is
6          entry Reset;
7          entry Increment;
8      end T;
9
10     task body T is
11         Cnt : Integer := 0;
12     begin
13         loop
14             select
15                 accept Reset do
16                     Cnt := 0;
17                 end Reset;
18                 Put_Line ("Reset");
19             or
20                 accept Increment do
21                     Cnt := Cnt + 1;
22                 end Increment;
23                 Put_Line ("In T's loop ("
24                     & Integer'Image (Cnt)
25                     & ")");
26             or
27                 terminate;
28             end select;
29         end loop;
30     end T;
31
32     begin
33         Put_Line ("In Main");
34
35         for I in 1 .. 4 loop
36             -- Calling T's entry multiple times
37             T.Increment;
38         end loop;
39
40         T.Reset;
41         for I in 1 .. 4 loop
42             -- Calling T's entry multiple times
43             T.Increment;
44         end loop;
45
46     end Show_Rendezvous_Loop;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Rendezvous_Loop
MD5: 0542dbc029cffb9f794d761bab9f3a9d

Runtime output

```

In Main
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)
In T's loop ( 4)
Reset
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)

```

(continues on next page)

(continued from previous page)

In T's loop (4)

In this example, the task body implements an infinite loop that accepts calls to the Reset and Increment entry. We make the following observations:

- The **accept** E **do** ... **end** block is used to increment a counter.
 - As long as task T is performing the **do** ... **end** block, the main task waits for the block to complete.
- The main task is calling the Increment entry multiple times in the loop from 1 .. 4. It is also calling the Reset entry before the second loop.
 - Because task T contains an infinite loop, it always accepts calls to the Reset and Increment entries.
 - When the master construct of the subtask (the Show_Rendezvous_Loop subprogram) completes, it checks the status of the T task. Even though task T could accept new calls to the Reset or Increment entries, the master construct is allowed to terminate task T due to the **or terminate** part of the **select** statement.

15.1.6 Cycling tasks

In a previous example, we saw how to delay a task a specified time by using the **delay** keyword. However, using delay statements in a loop is not enough to guarantee regular intervals between those delay statements. For example, we may have a call to a computationally intensive procedure between executions of successive delay statements:

```
while True loop
  delay 1.0;
  -- ^ Wait 1.0 seconds
  Computational_Intensive_App;
end loop;
```

In this case, we can't guarantee that exactly 10 seconds have elapsed after 10 calls to the delay statement because a time drift may be introduced by the Computational_Intensive_App procedure. In many cases, this time drift is not relevant, so using the **delay** keyword is good enough.

However, there are situations where a time drift isn't acceptable. In those cases, we need to use the **delay until** statement, which accepts a precise time for the end of the delay, allowing us to define a regular interval. This is useful, for example, in real-time applications.

We will soon see an example of how this time drift may be introduced and how the **delay until** statement circumvents the problem. But before we do that, we look at a package containing a procedure allowing us to measure the elapsed time (Show_Elapsed_Time) and a dummy Computational_Intensive_App procedure which is simulated by using a simple delay. This is the complete package:

Listing 214: delay_aux_pkg.ads

```
1 with Ada.Real_Time; use Ada.Real_Time;
2
3 package Delay_Aux_Pkg is
4
5   function Get_Start_Time return Time
6     with Inline;
7
8   procedure Show_Elapsed_Time
9     with Inline;
10
```

(continues on next page)

(continued from previous page)

```

11  procedure Computational_Intensive_App;
12  private
13      Start_Time    : Time := Clock;
14
15      function Get_Start_Time return Time is
16          (Start_Time);
17
18  end Delay_Aux_Pkg;

```

Listing 215: delay_aux_pkg.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Delay_Aux_Pkg is
4
5      procedure Show_Elapsed_Time is
6          Now_Time    : Time;
7          Elapsed_Time : Time_Span;
8      begin
9          Now_Time     := Clock;
10         Elapsed_Time := Now_Time - Start_Time;
11         Put_Line ("Elapsed time "
12                 & Duration'Image
13                 (To_Duration (Elapsed_Time))
14                 & " seconds");
15     end Show_Elapsed_Time;
16
17     procedure Computational_Intensive_App is
18     begin
19         delay 0.5;
20     end Computational_Intensive_App;
21
22 end Delay_Aux_Pkg;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: 422a38c1afa0bbd659ec81de88479e0a

Using this auxiliary package, we're now ready to write our time-drifting application:

Listing 216: show_time_task.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  with Delay_Aux_Pkg;
5
6  procedure Show_Time_Task is
7      package Aux renames Delay_Aux_Pkg;
8
9      task T;
10
11     task body T is
12         Cnt : Integer := 1;
13     begin
14         for I in 1 .. 5 loop
15             delay 1.0;
16
17             Aux.Show_Elapsed_Time;
18             Aux.Computational_Intensive_App;

```

(continues on next page)

(continued from previous page)

```

19
20     Put_Line ("Cycle # "
21             & Integer'Image (Cnt));
22     Cnt := Cnt + 1;
23 end loop;
24 Put_Line ("Finished time-drifting loop");
25 end T;
26
27 begin
28     null;
29 end Show_Time_Task;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: fe17c902fc127c0132677ea4005ff3f1

Runtime output

```

Elapsed time  1.000549316 seconds
Cycle #  1
Elapsed time  2.502470007 seconds
Cycle #  2
Elapsed time  4.003457739 seconds
Cycle #  3
Elapsed time  5.505746683 seconds
Cycle #  4
Elapsed time  7.006942816 seconds
Cycle #  5
Finished time-drifting loop

```

We can see by running the application that we already have a time difference of about four seconds after three iterations of the loop due to the drift introduced by Computational_Intensive_App. Using the **delay until** statement, however, we're able to avoid this time drift and have a regular interval of exactly one second:

Listing 217: show_time_task.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Ada.Real_Time;  use Ada.Real_Time;
3
4  with Delay_Aux_Pkg;
5
6  procedure Show_Time_Task is
7      package Aux renames Delay_Aux_Pkg;
8
9      task T;
10
11     task body T is
12         Cycle : constant Time_Span :=
13             Milliseconds (1000);
14         Next  : Time := Aux.Get_Start_Time
15             + Cycle;
16
17         Cnt   : Integer := 1;
18     begin
19         for I in 1 .. 5 loop
20             delay until Next;
21
22             Aux.Show_Elapsed_Time;
23             Aux.Computational_Intensive_App;

```

(continues on next page)

(continued from previous page)

```

24
25     -- Calculate next execution time
26     -- using a cycle of one second
27     Next := Next + Cycle;
28
29     Put_Line ("Cycle # "
30              & Integer'Image (Cnt));
31     Cnt := Cnt + 1;
32 end loop;
33 Put_Line ("Finished cycling");
34 end T;
35
36 begin
37     null;
38 end Show_Time_Task;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Time
MD5: 1456c0feee6def8b370d994c0ab75a15

Runtime output

```

Elapsed time  1.001883847 seconds
Cycle #  1
Elapsed time  2.001327181 seconds
Cycle #  2
Elapsed time  3.000699340 seconds
Cycle #  3
Elapsed time  4.001981741 seconds
Cycle #  4
Elapsed time  5.001690838 seconds
Cycle #  5
Finished cycling

```

Now, as we can see by running the application, the **delay until** statement ensures that the `Computational_Intensive_App` doesn't disturb the regular interval of one second between iterations.

15.2 Protected objects

When multiple tasks are accessing shared data, corruption of that data may occur. For example, data may be inconsistent if one task overwrites parts of the information that's being read by another task at the same time. In order to avoid these kinds of problems and ensure information is accessed in a coordinated way, we use *protected objects*.

Protected objects encapsulate data and provide access to that data by means of *protected operations*, which may be subprograms or protected entries. Using protected objects ensures that data is not corrupted by race conditions or other concurrent access.

Important

Objects can be protected from concurrent access using Ada tasks. In fact, this was the *only* way of protecting objects from concurrent access in Ada 83 (the first version of the Ada language). However, the use of protected objects is much simpler than using similar mechanisms implemented using only tasks. Therefore, you should use protected objects when your main goal is only to protect data.

15.2.1 Simple object

You declare a protected object with the **protected** keyword. The syntax is similar to that used for packages: you can declare operations (e.g., procedures and functions) in the public part and data in the private part. The corresponding implementation of the operations is included in the **protected body** of the object. For example:

Listing 218: show_protected_objects.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Protected_Objects is
4
5      protected Obj is
6          -- Operations go here (only subprograms)
7          procedure Set (V : Integer);
8          function Get return Integer;
9      private
10         -- Data goes here
11         Local : Integer := 0;
12     end Obj;
13
14     protected body Obj is
15         -- procedures can modify the data
16         procedure Set (V : Integer) is
17             begin
18                 Local := V;
19             end Set;
20
21         -- functions cannot modify the data
22         function Get return Integer is
23             begin
24                 return Local;
25             end Get;
26     end Obj;
27
28     begin
29         Obj.Set (5);
30         Put_Line ("Number is: "
31                 & Integer'Image (Obj.Get));
32     end Show_Protected_Objects;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Protected_Objects
MD5: dd97dd584ba2f13def3c04725d4e48a7

Runtime output

Number is: 5

In this example, we define two operations for `Obj`: `Set` and `Get`. The implementation of these operations is in the `Obj` body. The syntax used for writing these operations is the same as that for normal procedures and functions. The implementation of protected objects is straightforward — we simply access and update `Local` in these subprograms. To call these operations in the main application, we use prefixed notation, e.g., `Obj.Get`.

15.2.2 Entries

In addition to protected procedures and functions, you can also define protected entry points. Do this using the **entry** keyword. Protected entry points allow you to define barriers using the **when** keyword. Barriers are conditions that must be fulfilled before the entry can start performing its actual processing — we speak of *releasing* the barrier when the condition is fulfilled.

The previous example used procedures and functions to define operations on the protected objects. However, doing so permits reading protected information (via `Obj.Get`) before it's set (via `Obj.Set`). To allow that to be a defined operation, we specified a default value (0). Instead, by rewriting `Obj.Get` using an *entry* instead of a function, we implement a barrier, ensuring no task can read the information before it's been set.

The following example implements the barrier for the `Obj.Get` operation. It also contains two concurrent subprograms (main task and task T) that try to access the protected object.

Listing 219: `show_protected_objects_entries.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Protected_Objects_Entries is
4
5      protected Obj is
6          procedure Set (V : Integer);
7          entry Get (V : out Integer);
8      private
9          Local   : Integer;
10         Is_Set  : Boolean := False;
11     end Obj;
12
13     protected body Obj is
14         procedure Set (V : Integer) is
15             begin
16                 Local := V;
17                 Is_Set := True;
18             end Set;
19
20         entry Get (V : out Integer)
21             when Is_Set is
22             -- Entry is blocked until the
23             -- condition is true. The barrier
24             -- is evaluated at call of entries
25             -- and at exits of procedures and
26             -- entries. The calling task sleeps
27             -- until the barrier is released.
28             begin
29                 V := Local;
30                 Is_Set := False;
31             end Get;
32     end Obj;
33
34     N : Integer := 0;
35
36     task T;
37
38     task body T is
39     begin
40         Put_Line
41         ("Task T will delay for 4 seconds...");
42         delay 4.0;
43
44         Put_Line

```

(continues on next page)

(continued from previous page)

```

45     ("Task T will set Obj...");
46     Obj.Set (5);
47
48     Put_Line
49     ("Task T has just set Obj...");
50 end T;
51 begin
52     Put_Line
53     ("Main application will get Obj...");
54     Obj.Get (N);
55
56     Put_Line
57     ("Main application has retrieved Obj...");
58     Put_Line
59     ("Number is: " & Integer'Image (N));
60
61 end Show_Protected_Objects_Entries;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Protected_Objects_Entries
MD5: c1134445a96700b871fb76c4d6342359

Runtime output

```

Task T will delay for 4 seconds...
Main application will get Obj...
Task T will set Obj...
Task T has just set Obj...
Main application has retrieved Obj...
Number is: 5

```

As we see by running it, the main application waits until the protected object is set (by the call to `Obj.Set` in task T) before it reads the information (via `Obj.Get`). Because a 4-second delay has been added in task T, the main application is also delayed by 4 seconds. Only after this delay does task T set the object and release the barrier in `Obj.Get` so that the main application can then resume processing (after the information is retrieved from the protected object).

15.3 Task and protected types

In the previous examples, we defined single tasks and protected objects. We can, however, generalize tasks and protected objects using type definitions. This allows us, for example, to create multiple tasks based on just a single task type.

15.3.1 Task types

A task type is a generalization of a task. The declaration is similar to simple tasks: you replace **task** with **task type**. The difference between simple tasks and task types is that task types don't create actual tasks that automatically start. Instead, a task object declaration is needed. This is exactly the way normal variables and types work: objects are only created by variable definitions, not type definitions.

To illustrate this, we repeat our first example:

Listing 220: show_simple_task.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task is
4     task T;
5
6     task body T is
7     begin
8         Put_Line ("In task T");
9     end T;
10 begin
11     Put_Line ("In main");
12 end Show_Simple_Task;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Simple_Task
MD5: b17d9b35b4b2b53bc59776749e1be219

Runtime output

```

In task T
In main

```

We now rewrite it by replacing **task** T with **task type** TT. We declare a task (A_Task) based on the task type TT after its definition:

Listing 221: show_simple_task_type.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task_Type is
4     task type TT;
5
6     task body TT is
7     begin
8         Put_Line ("In task type TT");
9     end TT;
10
11     A_Task : TT;
12 begin
13     Put_Line ("In main");
14 end Show_Simple_Task_Type;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Simple_Task_Type
MD5: 24c26dcbbaf5c54f0a7d47c3c0da728

Runtime output

```

In task type TT
In main

```

We can extend this example and create an array of tasks. Since we're using the same syntax as for variable declarations, we use a similar syntax for task types: **array** (<=>) **of** **Task_Type**. Also, we can pass information to the individual tasks by defining a Start entry. Here's the updated example:

Listing 222: show_task_type_array.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Task_Type_Array is
4     task type TT is
5         entry Start (N : Integer);
6     end TT;
7
8     task body TT is
9         Task_N : Integer;
10    begin
11        accept Start (N : Integer) do
12            Task_N := N;
13        end Start;
14        Put_Line ("In task T: "
15                & Integer'Image (Task_N));
16    end TT;
17
18    My_Tasks : array (1 .. 5) of TT;
19 begin
20     Put_Line ("In main");
21
22     for I in My_Tasks'Range loop
23         My_Tasks (I).Start (I);
24     end loop;
25 end Show_Task_Type_Array;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Task_Type_Array
MD5: bba072dfc52fb2bfbe6e7b9f8191464

Runtime output

```

In main
In task T:  1
In task T:  2
In task T:  3
In task T:  4
In task T:  5

```

In this example, we're declaring five tasks in the array `My_Tasks`. We pass the array index to the individual tasks in the entry point (`Start`). After the synchronization between the individual subtasks and the main task, each subtask calls `Put_Line` concurrently.

15.3.2 Protected types

A protected type is a generalization of a protected object. The declaration is similar to that for protected objects: you replace **protected** with **protected type**. Like task types, protected types require an object declaration to create actual objects. Again, this is similar to variable declarations and allows for creating arrays (or other composite objects) of protected objects.

We can reuse a previous example and rewrite it to use a protected type:

Listing 223: show_protected_object_type.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Protected_Object_Type is

```

(continues on next page)

(continued from previous page)

```

4
5  protected type P_Obj_Type is
6      procedure Set (V : Integer);
7      function Get return Integer;
8  private
9      Local : Integer := 0;
10 end P_Obj_Type;
11
12 protected body P_Obj_Type is
13     procedure Set (V : Integer) is
14     begin
15         Local := V;
16     end Set;
17
18     function Get return Integer is
19     begin
20         return Local;
21     end Get;
22 end P_Obj_Type;
23
24 Obj : P_Obj_Type;
25 begin
26     Obj.Set (5);
27     Put_Line ("Number is: "
28              & Integer'Image (Obj.Get));
29 end Show_Protected_Object_Type;

```

Code block metadata

Project: Courses.Intro_To_Ada.Tasking.Show_Protected_Object_Type
MD5: c50321e55afef0d72f263fee0669e55f

Runtime output

Number is: 5

In this example, instead of directly defining the protected object `Obj`, we first define a protected type `P_Obj_Type` and then declare `Obj` as an object of that protected type. Note that the main application hasn't changed: we still use `Obj.Set` and `Obj.Get` to access the protected object, just like in the original example.

DESIGN BY CONTRACTS

Contracts are used in programming to codify expectations. Parameter modes of a subprogram can be viewed as a simple form of contracts. When the specification of subprogram `Op` declares a parameter using `in` mode, the caller of `Op` knows that the `in` argument won't be changed by `Op`. In other words, the caller expects that `Op` doesn't modify the argument it's providing, but just reads the information stored in the argument. Constraints and subtypes are other examples of contracts. In general, these specifications improve the consistency of the application.

Design-by-contract programming refers to techniques that include pre- and postconditions, subtype predicates, and type invariants. We study those topics in this chapter.

16.1 Pre- and postconditions

Pre- and postconditions provide expectations regarding input and output parameters of subprograms and return value of functions. If we say that certain requirements must be met before calling a subprogram `Op`, those are preconditions. Similarly, if certain requirements must be met after a call to the subprogram `Op`, those are postconditions. We can think of preconditions and postconditions as promises between the subprogram caller and the callee: a precondition is a promise from the caller to the callee, and a postcondition is a promise in the other direction.

Pre- and postconditions are specified using an aspect clause in the subprogram declaration. A `with Pre => <condition>` clause specifies a precondition and a `with Post => <condition>` clause specifies a postcondition.

The following code shows an example of preconditions:

Listing 224: `show_simple_precondition.adb`

```
1 procedure Show_Simple_Precondition is
2
3     procedure DB_Entry (Name : String;
4                         Age  : Natural)
5         with Pre => Name'Length > 0
6     is
7     begin
8         -- Missing implementation
9         null;
10    end DB_Entry;
11 begin
12     DB_Entry ("John", 30);
13
14     -- Precondition will fail!
15     DB_Entry ("", 21);
16 end Show_Simple_Precondition;
```

Code block metadata

Project: Courses.Intro_To_Ada.Contracts.Show_Simple_Precondition
MD5: 87b6e080555603111801a0fcd2469acd

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : failed precondition from show_simple_
↳ precondition.adb:5
```

In this example, we want to prevent the name field in our database from containing an empty string. We implement this requirement by using a precondition requiring that the length of the string used for the Name parameter of the DB_Entry procedure is greater than zero. If the DB_Entry procedure is called with an empty string for the Name parameter, the call will fail because the precondition is not met.

In the GNAT toolchain

GNAT handles pre- and postconditions by generating runtime assertions for them. By default, however, assertions aren't enabled. Therefore, in order to check pre- and postconditions at runtime, you need to enable assertions by using the *-gnata* switch.

Before we get to our next example, let's briefly discuss quantified expressions, which are quite useful in concisely writing pre- and postconditions. Quantified expressions return a Boolean value indicating whether elements of an array or container match the expected condition. They have the form: (**for all** I **in** A'Range => <condition on A(I)>), where A is an array and I is an index. Quantified expressions using **for all** check whether the condition is true for every element. For example:

```
(for all I in A'Range => A (I) = 0)
```

This quantified expression is only true when all elements of the array A have a value of zero.

Another kind of quantified expressions uses **for some**. The form looks similar: (**for some** I **in** A'Range => <condition on A(I)>). However, in this case the qualified expression tests whether the condition is true only on *some* elements (hence the name) instead of all elements.

We illustrate postconditions using the following example:

Listing 225: show_simple_postcondition.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Postcondition is
4
5     type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
6
7     type Int_8_Array is
8         array (Integer range <>) of Int_8;
9
10    function Square (A : Int_8) return Int_8 is
11        (A * A)
12        with Post => (if abs A in 0 | 1
13                     then Square'Result = abs A
14                     else Square'Result > A);
15
16    procedure Square (A : in out Int_8_Array)
17        with Post => (for all I in A'Range =>
18                     A (I) = A'Old (I) *
19                     A'Old (I))
```

(continues on next page)

(continued from previous page)

```

20  is
21  begin
22      for V of A loop
23          V := Square (V);
24      end loop;
25  end Square;
26
27  V : Int_8_Array := (-2, -1, 0, 1, 10, 11);
28  begin
29      for E of V loop
30          Put_Line ("Original: "
31                  & Int_8'Image (E));
32      end loop;
33      New_Line;
34
35      Square (V);
36      for E of V loop
37          Put_Line ("Square:   "
38                  & Int_8'Image (E));
39      end loop;
40  end Show_Simple_Postcondition;

```

Code block metadata

Project: Courses.Intro_To_Ada.Contracts.Show_Simple_Postcondition
MD5: b9bae9fe09cefcbe6769ad9cd6739e2a

Runtime output

```

Original: -2
Original: -1
Original:  0
Original:  1
Original: 10
Original: 11

Square:    4
Square:    1
Square:    0
Square:    1
Square:   100
Square:   121

```

We declare a signed 8-bit type `Int_8` and an array of that type (`Int_8_Array`). We want to ensure each element of the array is squared after calling the procedure `Square` for an object of the `Int_8_Array` type. We do this with a postcondition using a **for all** expression. This postcondition also uses the `'Old` attribute to refer to the original value of the parameter (before the call).

We also want to ensure that the result of calls to the `Square` function for the `Int_8` type are greater than the input to that call. To do that, we write a postcondition using the `'Result` attribute of the function and comparing it to the input value.

We can use both pre- and postconditions in the declaration of a single subprogram. For example:

Listing 226: show_simple_contract.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Simple_Contract is

```

(continues on next page)

(continued from previous page)

```

4
5  type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
6
7  function Square (A : Int_8) return Int_8 is
8      (A * A)
9      with
10         Pre => (Integer'Size >= Int_8'Size * 2
11                and Integer (A) *
12                        Integer (A) <=
13                        Integer (Int_8'Last)),
14         Post => (if abs A in 0 | 1
15                 then Square'Result = abs A
16                 else Square'Result > A);
17
18  V : Int_8;
19  begin
20  V := Square (11);
21  Put_Line ("Square of 11 is "
22           & Int_8'Image (V));
23
24  -- Precondition will fail...
25  V := Square (12);
26  Put_Line ("Square of 12 is "
27           & Int_8'Image (V));
28  end Show_Simple_Contract;

```

Code block metadata

Project: Courses.Intro_To_Ada.Contracts.Show_Simple_Contract
MD5: 1d928dd100704907c858562155f90ee2

Runtime output

```

Square of 11 is  121

raised ADA.Assertions.Assertion_Error : failed precondition from show_simple_
↳ contract.adb:10

```

In this example, we want to ensure that the input value of calls to the Square function for the Int_8 type won't cause overflow in that function. We do this by converting the input value to the **Integer** type, which is used for the temporary calculation, and check if the result is in the appropriate range for the Int_8 type. We have the same postcondition in this example as in the previous one.

16.2 Predicates

Predicates specify expectations regarding types. They're similar to pre- and postconditions, but apply to types instead of subprograms. Their conditions are checked for each object of a given type, which allows verifying that an object of type T is conformant to the requirements of its type.

There are two kinds of predicates: static and dynamic. In simple terms, static predicates are used to check objects at compile-time, while dynamic predicates are used for checks at run time. Normally, static predicates are used for scalar types and dynamic predicates for the more complex types.

Static and dynamic predicates are specified using the following clauses, respectively:

- **with** Static_Predicate => <property>
- **with** Dynamic_Predicate => <property>

Let's use the following example to illustrate dynamic predicates:

Listing 227: show_dynamic_predicate_courses.adb

```

1  with Ada.Calendar; use Ada.Calendar;
2
3  with Ada.Containers.Vectors;
4
5  with Ada.Strings.Unbounded;
6  use  Ada.Strings.Unbounded;
7
8  procedure Show_Dynamic_Predicate_Courses is
9
10     package Courses is
11         type Course_Container is private;
12
13         type Course is record
14             Name      : Unbounded_String;
15             Start_Date : Time;
16             End_Date  : Time;
17         end record
18         with Dynamic_Predicate =>
19             Course.Start_Date <= Course.End_Date;
20
21         procedure Add (CC : in out Course_Container;
22                      C  :      Course);
23     private
24         package Course_Vectors is new
25             Ada.Containers.Vectors
26             (Index_Type   => Natural,
27              Element_Type => Course);
28
29         type Course_Container is record
30             V : Course_Vectors.Vector;
31         end record;
32     end Courses;
33
34     package body Courses is
35         procedure Add (CC : in out Course_Container;
36                      C  :      Course) is
37         begin
38             CC.V.Append (C);
39         end Add;
40     end Courses;
41
42     use Courses;
43
44     CC : Course_Container;
45 begin
46     Add (CC,
47         Course'(
48             Name      =>
49                 To_Unbounded_String
50                 ("Intro to Photography"),
51             Start_Date =>
52                 Time_Of (2018, 5, 1),
53             End_Date   =>
54                 Time_Of (2018, 5, 10)));
55
56     -- This should trigger an error in the
57     -- dynamic predicate check
58     Add (CC,
59         Course'(

```

(continues on next page)

(continued from previous page)

```

60     Name      =>
61         To_Unbounded_String
62         ("Intro to Video Recording"),
63     Start_Date =>
64         Time_Of (2019, 5, 1),
65     End_Date   =>
66         Time_Of (2018, 5, 10));
67
68 end Show_Dynamic_Predicate_Courses;
```

Code block metadata

Project: Courses.Intro_To_Ada.Contracts.Show_Dynamic_Predicate_Courses
MD5: 8bd6539e72995fecefcdf9666ffd04f

Runtime output

```

raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_dynamic_
↳predicate_courses.adb:59
```

In this example, the package `Courses` defines a type `Course` and a type `Course_Container`, an object of which contains all courses. We want to ensure that the dates of each course are consistent, specifically that the start date is no later than the end date. To enforce this rule, we declare a dynamic predicate for the `Course` type that performs the check for each object. The predicate uses the type name where a variable of that type would normally be used: this is a reference to the instance of the object being tested.

Note that the example above makes use of unbounded strings and dates. Both types are available in Ada's standard library. Please refer to the following sections for more information about:

- the unbounded string type (`Unbounded_String`): [Unbounded Strings](#) (page 249) section;
- dates and times: [Dates & Times](#) (page 233) section.

Static predicates, as mentioned above, are mostly used for scalar types and checked during compilation. They're particularly useful for representing non-contiguous elements of an enumeration. A classic example is a list of week days:

```
type Week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

We can easily create a sub-list of work days in the week by specifying a **subtype** with a range based on `Week`. For example:

```
subtype Work_Week is Week range Mon .. Fri;
```

Ranges in Ada can only be specified as contiguous lists: they don't allow us to pick specific days. However, we may want to create a list containing just the first, middle and last day of the work week. To do that, we use a static predicate:

```

subtype Check_Days is Work_Week
  with Static_Predicate =>
    Check_Days in Mon | Wed | Fri;
```

Let's look at a complete example:

Listing 228: show_predicates.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Predicates is
4
5      type Week is (Mon, Tue, Wed, Thu,
6                    Fri, Sat, Sun);
7
8      subtype Work_Week is Week range Mon .. Fri;
9
10     subtype Test_Days is Work_Week
11     with Static_Predicate =>
12         Test_Days in Mon | Wed | Fri;
13
14     type Tests_Week is array (Week) of Natural
15     with Dynamic_Predicate =>
16         (for all I in Tests_Week'Range =>
17             (case I is
18                 when Test_Days =>
19                     Tests_Week (I) > 0,
20                 when others =>
21                     Tests_Week (I) = 0));
22
23     Num_Tests : Tests_Week :=
24         (Mon => 3, Tue => 0,
25          Wed => 4, Thu => 0,
26          Fri => 2, Sat => 0,
27          Sun => 0);
28
29     procedure Display_Tests (N : Tests_Week) is
30     begin
31         for I in Test_Days loop
32             Put_Line ("# tests on "
33                     & Test_Days'Image (I)
34                     & " => "
35                     & Integer'Image (N (I)));
36         end loop;
37     end Display_Tests;
38
39 begin
40     Display_Tests (Num_Tests);
41
42     -- Assigning non-conformant values to
43     -- individual elements of the Tests_Week
44     -- type does not trigger a predicate
45     -- check:
46     Num_Tests (Tue) := 2;
47
48     -- However, assignments with the "complete"
49     -- Tests_Week type trigger a predicate
50     -- check. For example:
51     --
52     -- Num_Tests := (others => 0);
53
54     -- Also, calling any subprogram with
55     -- parameters of Tests_Week type
56     -- triggers a predicate check. Therefore,
57     -- the following line will fail:
58     Display_Tests (Num_Tests);
59 end Show_Predicates;

```

Code block metadata


```
Project: Courses.Intro_To_Ada.Contracts.Show_Predicates
MD5: 126c47033fc67fc8b6d7f6479205e752
```

Runtime output

```
# tests on MON => 3
# tests on WED => 4
# tests on FRI => 2

raised ADA.Assertions.Assertion_Error : Dynamic_Predicate failed at show_
↳ predicates.adb:58
```

Here we have an application that wants to perform tests only on three days of the work week. These days are specified in the `Test_Days` subtype. We want to track the number of tests that occur each day. We declare the type `Tests_Week` as an array, an object of which will contain the number of tests done each day. According to our requirements, these tests should happen only in the aforementioned three days; on other days, no tests should be performed. This requirement is implemented with a dynamic predicate of the type `Tests_Week`. Finally, the actual information about these tests is stored in the array `Num_Tests`, which is an instance of the `Tests_Week` type.

The dynamic predicate of the `Tests_Week` type is verified during the initialization of `Num_Tests`. If we have a non-conformant value there, the check will fail. However, as we can see in our example, individual assignments to elements of the array do not trigger a check. We can't check for consistency at this point because the initialization of the a complex data structure (such as arrays or records) may not be performed with a single assignment. However, as soon as the object is passed as an argument to a subprogram, the dynamic predicate is checked because the subprogram requires the object to be consistent. This happens in the last call to `Display_Tests` in our example. Here, the predicate check fails because the previous assignment has a non-conformant value.

16.3 Type invariants

Type invariants are another way of specifying expectations regarding types. While predicates are used for *non-private* types, type invariants are used exclusively to define expectations about private types. If a type `T` from a package `P` has a type invariant, the results of operations on objects of type `T` are always consistent with that invariant.

Type invariants are specified with a **with** `Type_Invariant => <property>` clause. Like predicates, the *property* defines a condition that allows us to check if an object of type `T` is conformant to its requirements. In this sense, type invariants can be viewed as a sort of predicate for private types. However, there are some differences in terms of checks. The following table summarizes the differences:

Element	Subprogram parameter checks	Assignment checks
Predicates	On all in and out parameters	On assignments and explicit initializations
Type invariants	On out parameters returned from subprograms declared in the same public scope	On all initializations

We could rewrite our previous example and replace dynamic predicates by type invariants. It would look like this:

Listing 229: show_type_invariant.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Calendar; use Ada.Calendar;
3
4  with Ada.Containers.Vectors;
5
6  with Ada.Strings.Unbounded;
7  use Ada.Strings.Unbounded;
8
9  procedure Show_Type_Invariant is
10
11     package Courses is
12         type Course is private
13             with Type_Invariant => Check (Course);
14
15         type Course_Container is private;
16
17         procedure Add (CC : in out Course_Container;
18                      C : Course);
19
20         function Init
21             (Name : String;
22              Start_Date, End_Date : Time)
23             return Course;
24
25         function Check (C : Course)
26             return Boolean;
27
28     private
29         type Course is record
30             Name : Unbounded_String;
31             Start_Date : Time;
32             End_Date : Time;
33         end record;
34
35         function Check (C : Course)
36             return Boolean is
37             (C.Start_Date <= C.End_Date);
38
39         package Course_Vectors is new
40             Ada.Containers.Vectors
41             (Index_Type => Natural,
42              Element_Type => Course);
43
44         type Course_Container is record
45             V : Course_Vectors.Vector;
46         end record;
47     end Courses;
48
49     package body Courses is
50         procedure Add (CC : in out Course_Container;
51                      C : Course) is
52         begin
53             CC.V.Append (C);
54         end Add;
55
56         function Init
57             (Name : String;
58              Start_Date, End_Date : Time)
59             return Course is
60         begin
61             return

```

(continues on next page)

(continued from previous page)

```

62         Course'(Name      =>
63             To_Unbounded_String (Name),
64             Start_Date => Start_Date,
65             End_Date   => End_Date);
66     end Init;
67 end Courses;
68
69 use Courses;
70
71 CC : Course_Container;
72 begin
73     Add (CC,
74         Init (Name      =>
75             "Intro to Photography",
76             Start_Date =>
77                 Time_Of (2018, 5, 1),
78             End_Date   =>
79                 Time_Of (2018, 5, 10)));
80
81     -- This should trigger an error in the
82     -- type-invariant check
83     Add (CC,
84         Init (Name      =>
85             "Intro to Video Recording",
86             Start_Date =>
87                 Time_Of (2019, 5, 1),
88             End_Date   =>
89                 Time_Of (2018, 5, 10)));
90 end Show_Type_Invariant;

```

Code block metadata

Project: Courses.Intro_To_Ada.Contracts.Show_Type_Invariant
MD5: c6ef863da94285f927dd106645af8650

Runtime output

```

raised ADA.ASSERTIONS.ASSERTION_ERROR : failed invariant from show_type_invariant.
↪ adb:13

```

The major difference is that the Course type was a visible (public) type of the Courses package in the previous example, but in this example is a private type.

INTERFACING WITH C

Ada allows us to interface with code in many languages, including C and C++. This section discusses how to interface with C.

17.1 Multi-language project

By default, when using **gprbuild** we only compile Ada source files. To compile C files as well, we need to modify the project file used by **gprbuild**. We use the Languages entry, as in the following example:

```
project Multilang is
    for Languages use ("ada", "c");
    for Source_Dirs use ("src");
    for Main use ("main.adb");
    for Object_Dir use "obj";
end Multilang;
```

17.2 Type convention

To interface with data types declared in a C application, you specify the Convention aspect on the corresponding Ada type declaration. In the following example, we interface with the C_Enum enumeration declared in a C source file:

Listing 230: show_c_enum.adb

```
1 procedure Show_C_Enum is
2
3     type C_Enum is (A, B, C)
4         with Convention => C;
5     -- Use C convention for C_Enum
6 begin
7     null;
8 end Show_C_Enum;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Enum
MD5: a14d7d981fd7d6d806cf3c55f35e19c8
```

To interface with C's built-in types, we use the Interfaces.C package, which contains most of the type definitions we need. For example:

Listing 231: show_c_struct.adb

```
1 with Interfaces.C; use Interfaces.C;
2
3 procedure Show_C_Struct is
4
5     type c_struct is record
6         a : int;
7         b : long;
8         c : unsigned;
9         d : double;
10    end record
11    with Convention => C;
12
13 begin
14     null;
15 end Show_C_Struct;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Struct
MD5: dda4d3f8e4ddf5c5138a990a9a8ac427

Here, we're interfacing with a C struct (C_Struct) and using the corresponding data types in C (**int**, **long**, **unsigned** and **double**). This is the declaration in C:

Listing 232: c_struct.h

```
1 struct c_struct
2 {
3     int      a;
4     long     b;
5     unsigned c;
6     double   d;
7 };
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Struct
MD5: 58709b6a9eea2606d7ec0aaca0a749ff

17.3 Foreign subprograms

17.3.1 Calling C subprograms in Ada

We use a similar approach when interfacing with subprograms written in C. Consider the following declaration in the C header file:

Listing 233: my_func.h

```
1 int my_func (int a);
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 37b9d7ba668f7ec83c2b27ee33637937

Here's the corresponding C definition:

Listing 234: my_func.c

```

1  #include "my_func.h"
2
3  int my_func (int a)
4  {
5      return a * 2;
6  }

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 284b1639cb393fc14ed196d78429f3ba

We can interface this code in Ada using the Import aspect. For example:

Listing 235: show_c_func.adb

```

1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Show_C_Func is
5
6      function my_func (a : int) return int
7      with
8          Import      => True,
9          Convention  => C;
10
11      -- Imports function 'my_func' from C.
12      -- You can now call it from Ada.
13
14      V : int;
15  begin
16      V := my_func (2);
17      Put_Line ("Result is " & int'Image (V));
18  end Show_C_Func;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 6c5d85c1debdeaa642946eac413dfd2

If you want, you can use a different subprogram name in the Ada code. For example, we could call the C function Get_Value:

Listing 236: show_c_func.adb

```

1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Show_C_Func is
5
6      function Get_Value (a : int) return int
7      with
8          Import      => True,
9          Convention  => C,
10         External_Name => "my_func";
11
12      -- Imports function 'my_func' from C and
13      -- renames it to 'Get_Value'
14

```

(continues on next page)

(continued from previous page)

```
15     V : int;  
16 begin  
17     V := Get_Value (2);  
18     Put_Line ("Result is " & int'Image (V));  
19 end Show_C_Func;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Func
MD5: 856b4d99dfaa6946fb4597f254fd2f97

17.3.2 Calling Ada subprograms in C

You can also call Ada subprograms from C applications. You do this with the Export aspect. For example:

Listing 237: c_api.ads

```
1 with Interfaces.C; use Interfaces.C;  
2  
3 package C_API is  
4  
5     function My_Func (a : int) return int  
6     with  
7         Export      => True,  
8         Convention  => C,  
9         External_Name => "my_func";  
10  
11 end C_API;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Func
MD5: 00aa4ec29fc551e710900e2ee7d96bc9

This is the corresponding body that implements that function:

Listing 238: c_api.adb

```
1 package body C_API is  
2  
3     function My_Func (a : int) return int is  
4     begin  
5         return a * 2;  
6     end My_Func;  
7  
8 end C_API;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Func
MD5: 2b999ab431bbcleee223a654ad84b8248

On the C side, we do the same as we would if the function were written in C: simply declare it using the **extern** keyword. For example:

Listing 239: main.c

```
1 #include <stdio.h>  
2
```

(continues on next page)

(continued from previous page)

```

3 extern int my_func (int a);
4
5 int main (int argc, char **argv) {
6     int v = my_func(2);
7
8     printf("Result is %d\n", v);
9
10    return 0;
11 }
12

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Func
MD5: 69301036be9be16ed45895c2a86bc352

17.4 Foreign variables

17.4.1 Using C global variables in Ada

To use global variables from C code, we use the same method as subprograms: we specify the Import and Convention aspects for each variable we want to import.

Let's reuse an example from the previous section. We'll add a global variable (`func_cnt`) to count the number of times the function (`my_func`) is called:

Listing 240: test.h

```

1 extern int func_cnt;
2
3 int my_func (int a);

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Vars
MD5: 11ba8f7a72ce7058571994870a02b052

The variable is declared in the C file and incremented in `my_func`:

Listing 241: test.c

```

1 #include "test.h"
2
3 int func_cnt = 0;
4
5 int my_func (int a)
6 {
7     func_cnt++;
8
9     return a * 2;
10 }

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Vars
MD5: 23631537cb877a03d1243c94cb7b48e8

In the Ada application, we just reference the foreign variable:

Listing 242: show_c_func.adb

```

1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Show_C_Func is
5
6      function my_func (a : int) return int
7      with
8          Import      => True,
9          Convention => C;
10
11     V : int;
12
13     func_cnt : int
14     with
15         Import      => True,
16         Convention  => C;
17     -- We can access the func_cnt variable
18     -- from test.c
19
20     begin
21         V := my_func (1);
22         V := my_func (2);
23         V := my_func (3);
24
25         Put_Line ("Result is "
26                 & int'Image (V));
27
28         Put_Line ("Function was called "
29                 & int'Image (func_cnt)
30                 & " times");
31     end Show_C_Func;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.Ada_C_Vars
MD5: cf64a9dfbc6be853ba19729fe55f0ba4

As we see by running the application, the value of the counter is the number of times my_func was called.

We can use the External_Name aspect to give a different name for the variable in the Ada application in the same way we do for subprograms.

17.4.2 Using Ada variables in C

You can also use variables declared in Ada files in C applications. In the same way as we did for subprograms, you do this with the Export aspect.

Let's reuse a past example and add a counter, as in the previous example, but this time have the counter incremented in Ada code:

Listing 243: c_api.ads

```

1  with Interfaces.C; use Interfaces.C;
2
3  package C_API is
4
5      func_cnt : int := 0
6      with

```

(continues on next page)

(continued from previous page)

```

7      Export      => True,
8      Convention => C;
9
10     function My_Func (a : int) return int
11     with
12         Export      => True,
13         Convention   => C,
14         External_Name => "my_func";
15
16 end C_API;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Vars
MD5: fc118cddd797b669d2c68e57f90f69b2

The variable is then incremented in My_Func:

Listing 244: c_api.adb

```

1 package body C_API is
2
3     function My_Func (a : int) return int is
4     begin
5         func_cnt := func_cnt + 1;
6         return a * 2;
7     end My_Func;
8
9 end C_API;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Vars
MD5: adff5f3088da8b0dd853f1fb8b1e204f

In the C application, we just need to declare the variable and use it:

Listing 245: main.c

```

1 #include <stdio.h>
2
3 extern int my_func (int a);
4
5 extern int func_cnt;
6
7 int main (int argc, char **argv) {
8
9     int v;
10
11     v = my_func(1);
12     v = my_func(2);
13     v = my_func(3);
14
15     printf("Result is %d\n", v);
16
17     printf("Function was called %d times\n",
18         func_cnt);
19
20     return 0;
21 }

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Ada_Vars
MD5: 07fb3fbadb8ed4c0543fbfd7b5ef5c57

Again, by running the application, we see that the value from the counter is the number of times that `my_func` was called.

17.5 Generating bindings

In the examples above, we manually added aspects to our Ada code to correspond to the C source-code we're interfacing with. This is called creating a *binding*. We can automate this process by using the *Ada spec dump* compiler option: `-fdump-ada-spec`. We illustrate this by revisiting our previous example.

This was our C header file:

Listing 246: test.h

```
1 extern int func_cnt;
2
3 int my_func (int a);
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds
MD5: 11ba8f7a72ce7058571994870a02b052

To create Ada bindings, we'll call the compiler like this:

```
gcc -c -fdump-ada-spec -C ./test.h
```

The result is an Ada spec file called `test_h.ads`:

Listing 247: test_h.ads

```
1 pragma Ada_2005;
2 pragma Style_Checks (Off);
3
4 with Interfaces.C; use Interfaces.C;
5
6 package test_h is
7
8     func_cnt : aliased int; -- ./test.h:3
9     pragma Import (C, func_cnt, "func_cnt");
10
11     function my_func (arg1 : int) return int; -- ./test.h:5
12     pragma Import (C, my_func, "my_func");
13
14 end test_h;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds
MD5: 8d18aeae72dba3a9ab4f9f3943fab839

Now we simply refer to this `test_h` package in our Ada application:

Listing 248: show_c_func.adb

```
1 with Interfaces.C; use Interfaces.C;
2 with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```

3  with test_h;          use test_h;
4
5  procedure Show_C_Func is
6      V : int;
7  begin
8      V := my_func (1);
9      V := my_func (2);
10     V := my_func (3);
11
12     Put_Line ("Result is "
13              & int'Image (V));
14
15     Put_Line ("Function was called "
16              & int'Image (func_cnt)
17              & " times");
18 end Show_C_Func;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds
MD5: 8a07aae87b9f36c3fce84b75e8388933

You can specify the name of the parent unit for the bindings you're creating as the operand to fdump-ada-spec:

```
gcc -c -fdump-ada-spec -fada-spec-parent=Ext_C_Code -C ./test.h
```

This creates the file ext_c_code-test_h.ads:

Listing 249: ext_c_code-test_h.ads

```

1  package Ext_C_Code.test_h is
2
3      --  automatic generated bindings...
4
5  end Ext_C_Code.test_h;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_2
MD5: 3bd4087edff145a70d2a6db8543859ad

17.5.1 Adapting bindings

The compiler does the best it can when creating bindings for a C header file. However, sometimes it has to guess about the translation and the generated bindings don't always match our expectations. For example, this can happen when creating bindings for functions that have pointers as arguments. In this case, the compiler may use `System.Address` as the type of one or more pointers. Although this approach works fine (as we'll see later), this is usually not how a human would interpret the C header file. The following example illustrates this issue.

Let's start with this C header file:

Listing 250: test.h

```

1  struct test;
2
3  struct test * test_create(void);
4

```

(continues on next page)

(continued from previous page)

```

5 void test_destroy(struct test *t);
6
7 void test_reset(struct test *t);
8
9 void test_set_name(struct test *t,
10                  char *name);
11
12 void test_set_address(struct test *t,
13                     char *address);
14
15 void test_display(const struct test *t);

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: af642d9ea995bf01f13f8ff41bb0f4f6

And the corresponding C implementation:

Listing 251: test.c

```

1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4
5  #include "test.h"
6
7  struct test {
8      char name[80];
9      char address[120];
10 };
11
12 static size_t
13 strlcpy_stat(char *dst,
14             const char *src,
15             size_t dstsize)
16 {
17     size_t len = strlen(src);
18     if (dstsize) {
19         size_t bl = (len < dstsize-1 ?
20                     len : dstsize-1);
21         ((char*)memcpy(dst, src, bl))[bl] = 0;
22     }
23     return len;
24 }
25
26 struct test * test_create(void)
27 {
28     return malloc (sizeof (struct test));
29 }
30
31 void test_destroy(struct test *t)
32 {
33     if (t != NULL) {
34         free(t);
35     }
36 }
37
38 void test_reset(struct test *t)
39 {
40     t->name[0] = '\0';

```

(continues on next page)

(continued from previous page)

```

41  t->address[0] = '\0';
42  }
43
44  void test_set_name(struct test *t,
45                  char *name)
46  {
47      strcpy_stat(t->name,
48                  name,
49                  sizeof(t->name));
50  }
51
52  void test_set_address(struct test *t,
53                      char *address)
54  {
55      strcpy_stat(t->address,
56                  address,
57                  sizeof(t->address));
58  }
59
60  void test_display(const struct test *t)
61  {
62      printf("Name:   %s\n", t->name);
63      printf("Address: %s\n", t->address);
64  }

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 32652eb76ad92212609680d64e5687d3

Next, we'll create our bindings:

```
gcc -c -fdump-ada-spec -C ./test.h
```

This creates the following specification in test_h.ads:

Listing 252: test_h.ads

```

1  pragma Ada_2005;
2  pragma Style_Checks (Off);
3
4  with Interfaces.C; use Interfaces.C;
5  with System;
6  with Interfaces.C.Strings;
7
8  package test_h is
9
10     -- skipped empty struct test
11
12     function test_create return System.Address; -- ./test.h:5
13     pragma Import (C, test_create, "test_create");
14
15     procedure test_destroy (arg1 : System.Address); -- ./test.h:7
16     pragma Import (C, test_destroy, "test_destroy");
17
18     procedure test_reset (arg1 : System.Address); -- ./test.h:9
19     pragma Import (C, test_reset, "test_reset");
20
21     procedure test_set_name (arg1 : System.Address; arg2 : Interfaces.C.Strings.
22     ↪chars_ptr); -- ./test.h:11
23     pragma Import (C, test_set_name, "test_set_name");

```

(continues on next page)

(continued from previous page)

```

23
24     procedure test_set_address (arg1 : System.Address; arg2 : Interfaces.C.Strings.
↪chars_ptr); -- ./test.h:13
25     pragma Import (C, test_set_address, "test_set_address");
26
27     procedure test_display (arg1 : System.Address); -- ./test.h:15
28     pragma Import (C, test_display, "test_display");
29
30 end test_h;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 3bf8f01b94fd28594e4121a6a36afdf7

As we can see, the binding generator completely ignores the declaration `struct test` and all references to the test struct are replaced by addresses (`System.Address`). Nevertheless, these bindings are good enough to allow us to create a test application in Ada:

Listing 253: show_automatic_c_struct_bindings.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Interfaces.C;
4  use Interfaces.C;
5
6  with Interfaces.C.Strings;
7  use Interfaces.C.Strings;
8
9  with test_h; use test_h;
10
11 with System;
12
13 procedure Show_Automatic_C_Struct_Bindings is
14
15     Name      : constant chars_ptr :=
16         New_String ("John Doe");
17     Address   : constant chars_ptr :=
18         New_String ("Small Town");
19
20     T : System.Address := test_create;
21
22 begin
23     test_reset (T);
24     test_set_name (T, Name);
25     test_set_address (T, Address);
26
27     test_display (T);
28     test_destroy (T);
29 end Show_Automatic_C_Struct_Bindings;

```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 99d64fb14d9c869d140dd2fb7d3888d7

We can successfully bind our C code with Ada using the automatically-generated bindings, but they aren't ideal. Instead, we would prefer Ada bindings that match our (human) interpretation of the C header file. This requires manual analysis of the header file. The good news is that we can use the automatic generated bindings as a starting point and adapt them to our needs. For example, we can:

1. Define a Test type based on System.Address and use it in all relevant functions.
2. Remove the test_ prefix in all operations on the Test type.

This is the resulting specification:

Listing 254: adapted_test_h.ads

```

1  with System;
2
3  with Interfaces.C; use Interfaces.C;
4  with Interfaces.C.Strings;
5
6  package adapted_test_h is
7
8      type Test is new System.Address;
9
10     function Create return Test;
11     pragma Import (C, Create, "test_create");
12
13     procedure Destroy (T : Test);
14     pragma Import (C, Destroy, "test_destroy");
15
16     procedure Reset (T : Test);
17     pragma Import (C, Reset, "test_reset");
18
19     procedure Set_Name (T      : Test;
20                        Name    : Interfaces.C.Strings.chars_ptr); -- ./test.h:11
21     pragma Import (C, Set_Name, "test_set_name");
22
23     procedure Set_Address (T      : Test;
24                           Address : Interfaces.C.Strings.chars_ptr);
25     pragma Import (C, Set_Address, "test_set_address");
26
27     procedure Display (T : Test); -- ./test.h:15
28     pragma Import (C, Display, "test_display");
29
30 end adapted_test_h;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 5cc875e1b01af839141e5e623f6c5b7a

And this is the corresponding Ada body:

Listing 255: show_adapted_c_struct_bindings.adb

```

1  with Interfaces.C;
2  use Interfaces.C;
3
4  with Interfaces.C.Strings;
5  use Interfaces.C.Strings;
6
7  with adapted_test_h; use adapted_test_h;
8
9  with System;
10
11 procedure Show_Adapted_C_Struct_Bindings is
12
13     Name      : constant chars_ptr :=
14         New_String ("John Doe");
15     Address   : constant chars_ptr :=
```

(continues on next page)

(continued from previous page)

```
16     New_String ("Small Town");
17
18     T : Test := Create;
19
20 begin
21     Reset (T);
22     Set_Name (T, Name);
23     Set_Address (T, Address);
24
25     Display (T);
26     Destroy (T);
27 end Show_Adapted_C_Struct_Bindings;
```

Code block metadata

Project: Courses.Intro_To_Ada.Interfacing_With_C.C_Binds_3
MD5: 626d07b080fbbd2bf1d5f9140b64955c

Now we can use the Test type and its operations in a clean, readable way.

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a large and ill-defined concept in programming languages and one that tends to encompass many different meanings because different languages often implement their own vision of it, with similarities and differences from the implementations in other languages.

However, one model mostly "won" the battle of what object-oriented means, if only by sheer popularity. It's the model used in the Java programming language, which is very similar to the one used by C++. Here are some defining characteristics:

- Type derivation and extension: Most object oriented languages allow the user to add fields to derived types.
- Subtyping: Objects of a type derived from a base type can, in some instances, be substituted for objects of the base type.
- Runtime polymorphism: Calling a subprogram, usually called a *method*, attached to an object type can dispatch at runtime depending on the exact type of the object.
- Encapsulation: Objects can hide some of their data.
- Extensibility: People from the "outside" of your package, or even your whole library, can derive from your object types and define their own behaviors.

Ada dates from before object-oriented programming was as popular as it is today. Some of the mechanisms and concepts from the above list were in the earliest version of Ada even before what we would call OOP was added:

- As we saw, encapsulation is not implemented at the type level in Ada, but instead at the package level.
- Subtyping can be implemented using, well, subtypes, which have a full and permissive static substitutability model. The substitution will fail at runtime if the dynamic constraints of the subtype are not fulfilled.
- Runtime polymorphism can be implemented using variant records.

However, this lists leaves out type extensions, if you don't consider variant records, and extensibility.

The 1995 revision of Ada added a feature filling the gaps, which allowed people to program following the object-oriented paradigm in an easier fashion. This feature is called *tagged types*.

Note

It's possible to program in Ada without ever creating tagged types. If that's your preferred style of programming or you have no specific use for tagged types, feel free to not use them, as is the case for many features of Ada.

However, they can be the best way to express solutions to certain problems and they may be the best way to solve your problem. If that's the case, read on!

18.1 Derived types

Before presenting tagged types, we should discuss a topic we have brushed on, but not really covered, up to now:

You can create one or more new types from every type in Ada. Type derivation is built into the language.

Listing 256: newtypes.ads

```

1 package Newtypes is
2   type Point is record
3     X, Y : Integer;
4   end record;
5
6   type New_Point is new Point;
7 end Newtypes;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Newtypes
MD5: 0d45096755b4bfb08ba8db19ecba3f57

Type derivation is useful to enforce strong typing because the type system treats the two types as incompatible.

But the benefits are not limited to that: you can inherit things from the type you derive from. You not only inherit the representation of the data, but you can also inherit behavior.

When you inherit a type you also inherit what are called *primitive operations*. A primitive operation (or just a *primitive*) is a subprogram attached to a type. Ada defines primitives as subprograms defined in the same scope as the type.

Attention

A subprogram will only become a primitive of the type if:

1. The subprogram is declared in the same scope as the type and
2. The type and the subprogram are declared in a package

Listing 257: primitives.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Primitives is
4   package Week is
5     type Days is (Monday, Tuesday, Wednesday,
6                  Thursday, Friday,
```

(continues on next page)

(continued from previous page)

```

7         Saturday, Sunday));
8
9     -- Print_Day is a primitive
10    -- of the type Days
11    procedure Print_Day (D : Days);
12    end Week;
13
14    package body Week is
15        procedure Print_Day (D : Days) is
16        begin
17            Put_Line (Days'Image (D));
18        end Print_Day;
19    end Week;
20
21    use Week;
22    type Weekend_Days is new
23        Days range Saturday .. Sunday;
24
25    -- A procedure Print_Day is automatically
26    -- inherited here. It is as if the procedure
27    --
28    -- procedure Print_Day (D : Weekend_Days);
29    --
30    -- has been declared with the same body
31
32    Sat : Weekend_Days := Saturday;
33    begin
34        Print_Day (Sat);
35    end Primitives;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Primitives
MD5: eb1b0eb66f03a4a17bd9686ec4e12e2e

Runtime output

SATURDAY

This kind of inheritance can be very useful, and is not limited to record types (you can use it on discrete types, as in the example above), but it's only superficially similar to object-oriented inheritance:

- Records can't be extended using this mechanism alone. You also can't specify a new representation for the new type: it will **always** have the same representation as the base type.
- There's no facility for dynamic dispatch or polymorphism. Objects are of a fixed, static type.

There are other differences, but it's not useful to list them all here. Just remember that this is a kind of inheritance you can use if you only want to statically inherit behavior without duplicating code or using composition, but a kind you can't use if you want any dynamic features that are usually associated with OOP.

18.2 Tagged types

The 1995 revision of the Ada language introduced tagged types to fulfil the need for an unified solution that allows programming in an object-oriented style similar to the one described at the beginning of this chapter.

Tagged types are very similar to normal records except that some functionality is added:

- Types have a *tag*, stored inside each object, that identifies the [runtime type](#)¹⁹ of that object.
- Primitives can dispatch. A primitive on a tagged type is what you would call a *method* in Java or C++. If you derive a base type and override a primitive of it, you can often call it on an object with the result that which primitive is called depends on the exact runtime type of the object.
- Subtyping rules are introduced allowing a tagged type derived from a base type to be statically compatible with the base type.

Let's see our first tagged type declarations:

Listing 258: p.ads

```
1 package P is
2   type My_Class is tagged null record;
3   -- Just like a regular record, but
4   -- with tagged qualifier
5
6   -- Methods are outside of the type
7   -- definition:
8
9   procedure Foo (Self : in out My_Class);
10  -- If you define a procedure taking a
11  -- My_Class argument in the same package,
12  -- it will be a method.
13
14  -- Here's how you derive a tagged type:
15
16  type Derived is new My_Class with record
17    A : Integer;
18    -- You can add fields in derived types.
19  end record;
20
21  overriding
22  procedure Foo (Self : in out Derived);
23  -- The "overriding" qualifier is optional,
24  -- but if it is present, it must be valid.
25 end P;
```

Listing 259: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4   procedure Foo (Self : in out My_Class) is
5   begin
6     Put_Line ("In My_Class.Foo");
7   end Foo;
8
9   procedure Foo (Self : in out Derived) is
10  begin
11    Put_Line ("In Derived.Foo, A = "
12              & Integer'Image (Self.A));
13  end Foo;
14 end P;
```

Code block metadata

¹⁹ https://en.wikipedia.org/wiki/Run-time_type_information

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
 MD5: 45baaad66a1047358addb574d0fa00bc

18.3 Classwide types

To remain consistent with the rest of the language, a new notation needed to be introduced to say "This object is of this type or any descendant derives tagged type".

In Ada, we call this the *classwide type*. It's used in OOP as soon as you need polymorphism. For example, you can't do the following:

Listing 260: main.adb

```

1  with P; use P;
2
3  procedure Main is
4
5      01 : My_Class;
6      -- Declaring an object of type My_Class
7
8      02 : Derived := (A => 12);
9      -- Declaring an object of type Derived
10
11     03 : My_Class := 02;
12     -- INVALID: Trying to assign a value
13     -- of type derived to a variable of
14     -- type My_Class.
15 begin
16     null;
17 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
 MD5: c87ad8bb686cb1763740750846258357

Build output

```

main.adb:11:21: error: expected type "My_Class" defined at p.ads:2
main.adb:11:21: error: found type "Derived" defined at p.ads:16
gprbuild: *** compilation phase failed
```

This is because an object of a type T is exactly of the type T, whether T is tagged or not. What you want to say as a programmer is "I want O3 to be able to hold an object of type My_Class or any type descending from My_Class". Here's how you do that:

Listing 261: main.adb

```

1  with P; use P;
2
3  procedure Main is
4
5      01 : My_Class;
6      -- Declare an object of type My_Class
7
8      02 : Derived := (A => 12);
9      -- Declare an object of type Derived
10
11     03 : My_Class'Class := 02;
12     -- Now valid: My_Class'Class designates
13     -- the classwide type for My_Class,
```

(continues on next page)

(continued from previous page)

```

13  -- which is the set of all types
14  -- descending from My_Class (including
15  -- My_Class).
16  begin
17    null;
18  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: 35412176a248015a26e507164ce526af

Attention

Because an object of a classwide type can be the size of any descendant of its base type, it has an unknown size. It's therefore an indefinite type, with the expected restrictions:

- It can't be stored as a field/component of a record
- An object of a classwide type needs to be initialized immediately (you can't specify the constraints of such a type in any way other than by initializing it).

18.4 Dispatching operations

We saw that you can override operations in types derived from another tagged type. The eventual goal of OOP is to make a dispatching call: a call to a primitive (method) that depends on the exact type of the object.

But, if you think carefully about it, a variable of type `My_Class` always contains an object of exactly that type. If you want to have a variable that can contain a `My_Class` or any derived type, it has to be of type `My_Class'Class`.

In other words, to make a dispatching call, you must first have an object that can be either of a type or any type derived from this type, namely an object of a classwide type.

Listing 262: main.adb

```

1  with P; use P;
2
3  procedure Main is
4    01 : My_Class;
5    -- Declare an object of type My_Class
6
7    02 : Derived := (A => 12);
8    -- Declare an object of type Derived
9
10   03 : My_Class'Class := 02;
11
12   04 : My_Class'Class := 01;
13  begin
14    Foo (01);
15    -- Non dispatching: Calls My_Class.Foo
16    Foo (02);
17    -- Non dispatching: Calls Derived.Foo
18    Foo (03);
19    -- Dispatching: Calls Derived.Foo
20    Foo (04);

```

(continues on next page)

(continued from previous page)

```

21  -- Dispatching: Calls My_Class.Foo
22  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: 7631f823b0dd9e5474f6bb2dc35af2a2

Runtime output

```

In My_Class.Foo
In Derived.Foo, A = 12
In Derived.Foo, A = 12
In My_Class.Foo

```

Attention

You can convert an object of type `Derived` to an object of type `My_Class`. This is called a *view conversion* in Ada parlance and is useful, for example, if you want to call a parent method.

In that case, the object really is converted to a `My_Class` object, which means its tag is changed. Since tagged objects are always passed by reference, you can use this kind of conversion to modify the state of an object: changes to converted object will affect the original one.

Listing 263: main.adb

```

1  with P; use P;
2
3  procedure Main is
4      01 : Derived := (A => 12);
5      -- Declare an object of type Derived
6
7      02 : My_Class := My_Class (01);
8
9      03 : My_Class'Class := 02;
10  begin
11      Foo (01);
12      -- Non dispatching: Calls Derived.Foo
13      Foo (02);
14      -- Non dispatching: Calls My_Class.Foo
15
16      Foo (03);
17      -- Dispatching: Calls My_Class.Foo
18  end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: b92112b05201ff14789baca258fa0cbc

Runtime output

```

In Derived.Foo, A = 12
In My_Class.Foo
In My_Class.Foo

```


18.5 Dot notation

You can also call primitives of tagged types with a notation that's more familiar to object oriented programmers. Given the `Foo` primitive above, you can also write the above program this way:

Listing 264: main.adb

```

1  with P; use P;
2
3  procedure Main is
4      01 : My_Class;
5      --  Declare an object of type My_Class
6
7      02 : Derived := (A => 12);
8      --  Declare an object of type Derived
9
10     03 : My_Class'Class := 02;
11
12     04 : My_Class'Class := 01;
13 begin
14     01.Foo;
15     --  Non dispatching: Calls My_Class.Foo
16     02.Foo;
17     --  Non dispatching: Calls Derived.Foo
18     03.Foo;
19     --  Dispatching: Calls Derived.Foo
20     04.Foo;
21     --  Dispatching: Calls My_Class.Foo
22 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: 9c6ebdfec9ceeb986d92eb90ec9ff59b

Runtime output

```

In My_Class.Foo
In Derived.Foo, A = 12
In Derived.Foo, A = 12
In My_Class.Foo

```

If the dispatching parameter of a primitive is the first parameter, which is the case in our examples, you can call the primitive using the dot notation. Any remaining parameter are passed normally:

Listing 265: main.adb

```

1  with P; use P;
2
3  procedure Main is
4      package Extend is
5          type D2 is new Derived with null record;
6
7          procedure Bar (Self : in out D2;
8                        Val  : Integer);
9      end Extend;
10
11     package body Extend is
12         procedure Bar (Self : in out D2;

```

(continues on next page)

(continued from previous page)

```

13         Val : Integer) is
14     begin
15         Self.A := Self.A + Val;
16     end Bar;
17 end Extend;
18
19 use Extend;
20
21 Obj : D2 := (A => 15);
22 begin
23     Obj.Bar (2);
24     Obj.Foo;
25 end Main;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Types
MD5: fec4f5cc4213cc111708dcc276e870c2

Runtime output

In Derived.Foo, A = 17

18.6 Private & Limited

We've seen previously (in the *Privacy* (page 115) chapter) that types can be declared limited or private. These encapsulation techniques can also be applied to tagged types, as we'll see in this section.

This is an example of a tagged private type:

Listing 266: p.ads

```

1 package P is
2     type T is tagged private;
3 private
4     type T is tagged record
5         E : Integer;
6     end record;
7 end P;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Private_Types
MD5: 4cd4bcd1a54d5f6407a500558b5da417

This is an example of a tagged limited type:

Listing 267: p.ads

```

1 package P is
2     type T is tagged limited record
3         E : Integer;
4     end record;
5 end P;

```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Limited_Types
MD5: 13228777133aadb97da1c29f732459c

Naturally, you can combine both *limited* and *private* types and declare a tagged limited private type:

Listing 268: p.ads

```
1 package P is
2   type T is tagged limited private;
3
4   procedure Init (A : in out T);
5 private
6   type T is tagged limited record
7     E : Integer;
8   end record;
9 end P;
```

Listing 269: p.adb

```
1 package body P is
2
3   procedure Init (A : in out T) is
4   begin
5     A.E := 0;
6   end Init;
7
8 end P;
```

Listing 270: main.adb

```
1 with P; use P;
2
3 procedure Main is
4   T1, T2 : T;
5 begin
6   T1.Init;
7   T2.Init;
8
9   -- The following line doesn't work
10  -- because type T is private:
11  --
12  -- T1.E := 0;
13
14  -- The following line doesn't work
15  -- because type T is limited:
16  --
17  -- T2 := T1;
18 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Tagged_Limited_Private_Types
MD5: 68240374505bcaf7aad4ebaed3b9127b

Note that the code in the Main procedure above presents two assignments that trigger compilation errors because type T is limited private. In fact, you cannot:

- assign to T1.E directly because type T is private;
- assign T1 to T2 because type T is limited.

In this case, there's no distinction between tagged and non-tagged types: these compilation errors would also occur for non-tagged types.

18.7 Classwide access types

In this section, we'll discuss an useful pattern for object-oriented programming in Ada: classwide access type. Let's start with an example where we declare a tagged type `T` and a derived type `T_New`:

Listing 271: p.ads

```

1 package P is
2   type T is tagged null record;
3
4   procedure Show (Dummy : T);
5
6   type T_New is new T with null record;
7
8   procedure Show (Dummy : T_New);
9 end P;
```

Listing 272: p.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5   procedure Show (Dummy : T) is
6   begin
7     Put_Line ("Using type "
8               & T'External_Tag);
9   end Show;
10
11  procedure Show (Dummy : T_New) is
12  begin
13    Put_Line ("Using type "
14              & T_New'External_Tag);
15  end Show;
16
17 end P;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: fd5cb99925d3c88536546aa0be8104b7

Note that we're using null records for both types `T` and `T_New`. Although these types don't actually have any component, we can still use them to demonstrate dispatching. Also note that the example above makes use of the `'External_Tag` attribute in the implementation of the `Show` procedure to get a string for the corresponding tagged type.

As we've seen before, we must use a classwide type to create objects that can make dispatching calls. In other words, objects of type `T'Class` will dispatch. For example:

Listing 273: dispatching_example.adb

```

1 with P; use P;
2
3 procedure Dispatching_Example is
4   T2      : T_New;
5   T_Dispatch : constant T'Class := T2;
6 begin
7   T_Dispatch.Show;
8 end Dispatching_Example;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: f8957b31c9c62db23759baad7b867a57

Runtime output

Using type P.T_NEW

A more useful application is to declare an array of objects that can dispatch. For example, we'd like to declare an array T_Arr, loop over this array and dispatch according to the actual type of each individual element:

```
for I in T_Arr'Range loop
  T_Arr (I).Show;
  -- Call Show procedure according
  -- to actual type of T_Arr (I)
end loop;
```

However, it's not possible to declare an array of type T'Class directly:

Listing 274: classwide_compilation_error.adb

```
1 with P; use P;
2
3 procedure Classwide_Compilation_Error is
4   T_Arr : array (1 .. 2) of T'Class;
5   --
6   --           ^
7   --           Compilation Error!
8 begin
9   for I in T_Arr'Range loop
10    T_Arr (I).Show;
11  end loop;
end Classwide_Compilation_Error;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Error
MD5: e86f6c6ee35dced8f330bf6177d178fd

Build output

```
classwide_compilation_error.adb:4:32: error: unconstrained element type in array_
↳ declaration
gprbuild: *** compilation phase failed
```

In fact, it's impossible for the compiler to know which type would actually be used for each element of the array. However, if we use dynamic allocation via access types, we can allocate objects of different types for the individual elements of an array T_Arr. We do this by using classwide access types, which have the following format:

```
type T_Class is access T'Class;
```

We can rewrite the previous example using the T_Class type. In this case, dynamically allocated objects of this type will dispatch according to the actual type used during the allocation. Also, let's introduce an Init procedure that won't be overridden for the derived T_New type. This is the adapted code:

Listing 275: p.ads

```
1 package P is
2   type T is tagged record
```

(continues on next page)

(continued from previous page)

```

3      E : Integer;
4  end record;
5
6  type T_Class is access T'Class;
7
8  procedure Init (A : in out T);
9
10 procedure Show (Dummy : T);
11
12 type T_New is new T with null record;
13
14 procedure Show (Dummy : T_New);
15
16 end P;
```

Listing 276: p.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
4
5      procedure Init (A : in out T) is
6      begin
7          Put_Line ("Initializing type T...");
8          A.E := 0;
9      end Init;
10
11     procedure Show (Dummy : T) is
12     begin
13         Put_Line ("Using type "
14                 & T'External_Tag);
15     end Show;
16
17     procedure Show (Dummy : T_New) is
18     begin
19         Put_Line ("Using type "
20                 & T_New'External_Tag);
21     end Show;
22
23 end P;
```

Listing 277: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with P;           use P;
3
4  procedure Main is
5      T_Arr : array (1 .. 2) of T_Class;
6  begin
7      T_Arr (1) := new T;
8      T_Arr (2) := new T_New;
9
10     for I in T_Arr'Range loop
11         Put_Line ("Element # "
12                 & Integer'Image (I));
13
14         T_Arr (I).Init;
15         T_Arr (I).Show;
16
17         Put_Line ("-----");
```

(continues on next page)

(continued from previous page)

```
18   end loop;  
19 end Main;
```

Code block metadata

Project: Courses.Intro_To_Ada.Object_Oriented_Programming.Classwide_Access
MD5: 97c05a8f911d0a0e39c0cc90fae184a7

Runtime output

```
Element # 1  
Initializing type T...  
Using type P.T  
-----  
Element # 2  
Initializing type T...  
Using type P.T_NEW  
-----
```

In this example, the first element (`T_Arr (1)`) is of type `T`, while the second element is of type `T_New`. When running the example, the `Init` procedure of type `T` is called for both elements of the `T_Arr` array, while the call to the `Show` procedure selects the corresponding procedure according to the type of each element of `T_Arr`.

STANDARD LIBRARY: CONTAINERS

In previous chapters, we've used arrays as the standard way to group multiple objects of a specific data type. In many cases, arrays are good enough for manipulating those objects. However, there are situations that require more flexibility and more advanced operations. For those cases, Ada provides support for containers — such as vectors and sets — in its standard library.

We present an introduction to containers here. For a list of all containers available in Ada, see [Appendix B](#) (page 273).

19.1 Vectors

In the following sections, we present a general overview of vectors, including instantiation, initialization, and operations on vector elements and vectors.

19.1.1 Instantiation

Here's an example showing the instantiation and declaration of a vector V:

Listing 278: show_vector_inst.adb

```
1 with Ada.Containers.Vectors;  
2  
3 procedure Show_Vector_Inst is  
4  
5     package Integer_Vectors is new  
6         Ada.Containers.Vectors  
7         (Index_Type => Natural,  
8          Element_Type => Integer);  
9  
10    V : Integer_Vectors.Vector;  
11 begin  
12     null;  
13 end Show_Vector_Inst;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Inst
MD5: 8b737842d2784f25502990f21e1cf6de

Containers are based on generic packages, so we can't simply declare a vector as we would declare an array of a specific type:

```
A : array (1 .. 10) of Integer;
```

Instead, we first need to instantiate one of those packages. We **with** the container package (Ada.Containers.Vectors in this case) and instantiate it to create an instance of the

generic package for the desired type. Only then can we declare the vector using the type from the instantiated package. This instantiation needs to be done for any container type from the standard library.

In the instantiation of `Integer_Vectors`, we indicate that the vector contains elements of **Integer** type by specifying it as the `Element_Type`. By setting `Index_Type` to **Natural**, we specify that the allowed range includes all natural numbers. We could have used a more restrictive range if desired.

19.1.2 Initialization

One way to initialize a vector is from a concatenation of elements. We use the `&` operator, as shown in the following example:

Listing 279: `show_vector_init.adb`

```
1 with Ada.Containers; use Ada.Containers;
2 with Ada.Containers.Vectors;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Vector_Init is
7
8     package Integer_Vectors is new
9         Ada.Containers.Vectors
10         (Index_Type => Natural,
11          Element_Type => Integer);
12
13     use Integer_Vectors;
14
15     V : Vector := 20 & 10 & 0 & 13;
16 begin
17     Put_Line ("Vector has "
18              & Count_Type'Image (V.Length)
19              & " elements");
20 end Show_Vector_Init;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Standard_Library.Show_Vector_Init`
MD5: `0087b0a15e0c88b27ac36c3b27159a17`

Runtime output

Vector has 4 elements

We specify `use Integer_Vectors`, so we have direct access to the types and operations from the instantiated package. Also, the example introduces another operation on the vector: `Length`, which retrieves the number of elements in the vector. We can use the dot notation because `Vector` is a tagged type, allowing us to write either `V.Length` or `Length (V)`.

19.1.3 Appending and prepending elements

You add elements to a vector using the `Prepend` and `Append` operations. As the names suggest, these operations add elements to the beginning or end of a vector, respectively. For example:

Listing 280: show_vector_append.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Append is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      use Integer_Vectors;
14
15      V : Vector;
16  begin
17      Put_Line ("Appending some elements "
18              & "to the vector...");
19      V.Append (20);
20      V.Append (10);
21      V.Append (0);
22      V.Append (13);
23      Put_Line ("Finished appending.");
24
25      Put_Line ("Prepending some elements"
26              & "to the vector...");
27      V.Prepend (30);
28      V.Prepend (40);
29      V.Prepend (100);
30      Put_Line ("Finished prepending.");
31
32      Put_Line ("Vector has "
33              & Count_Type'Image (V.Length)
34              & " elements");
35  end Show_Vector_Append;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Append
MD5: f88d393ba96a7950f58d9f1c0c74a021

Runtime output

```

Appending some elements to the vector...
Finished appending.
Prepending some elementsto the vector...
Finished prepending.
Vector has  7 elements

```

This example puts elements into the vector in the following sequence: (100, 40, 30, 20, 10, 0, 13).

The Reference Manual specifies that the worst-case complexity must be:

- $O(\log N)$ for the Append operation, and
- $O(N \log N)$ for the Prepend operation.

19.1.4 Accessing first and last elements

We access the first and last elements of a vector using the `First_Element` and `Last_Element` functions. For example:

Listing 281: `show_vector_first_last_element.adb`

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_First_Last_Element is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      use Integer_Vectors;
14
15      function Img (I : Integer)    return String
16          renames Integer'Image;
17      function Img (I : Count_Type) return String
18          renames Count_Type'Image;
19
20      V : Vector := 20 & 10 & 0 & 13;
21  begin
22      Put_Line ("Vector has "
23              & Img (V.Length)
24              & " elements");
25
26      -- Using V.First_Element to
27      -- retrieve first element
28      Put_Line ("First element is "
29              & Img (V.First_Element));
30
31      -- Using V.Last_Element to
32      -- retrieve last element
33      Put_Line ("Last element is "
34              & Img (V.Last_Element));
35  end Show_Vector_First_Last_Element;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_First_Last_Element
 MD5: 602255760d0017ced6b4115c845cd48d

Runtime output

```

Vector has  4 elements
First element is  20
Last element is  13
```

You can swap elements by calling the procedure `Swap` and retrieving a reference (a *cursor*) to the first and last elements of the vector by calling `First` and `Last`. A cursor allows us to iterate over a container and process individual elements from it.

With these operations, we're able to write code to swap the first and last elements of a vector:

Listing 282: show_vector_first_last_element.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_First_Last_Element is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      use Integer_Vectors;
14
15      function Img (I : Integer) return String
16          renames Integer'Image;
17
18      V : Vector := 20 & 10 & 0 & 13;
19  begin
20      -- We use V.First and V.Last to retrieve
21      -- cursor for first and last elements.
22      -- We use V.Swap to swap elements.
23      V.Swap (V.First, V.Last);
24
25      Put_Line ("First element is now "
26              & Img (V.First_Element));
27      Put_Line ("Last element is now "
28              & Img (V.Last_Element));
29  end Show_Vector_First_Last_Element;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_First_Last_Element
MD5: 1a0c0bf28bb661b3f328473ac3c2eb54

Runtime output

```

First element is now 13
Last element is now 20

```

19.1.5 Iterating

The easiest way to iterate over a container is to use a **for E of** Our_Container loop. This gives us a reference (E) to the element at the current position. We can then use E directly. For example:

Listing 283: show_vector_iteration.adb

```

1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Vector_Iteration is
6
7      package Integer_Vectors is new
8          Ada.Containers.Vectors
9             (Index_Type => Natural,
10              Element_Type => Integer);

```

(continues on next page)

(continued from previous page)

```

11
12     use Integer_Vectors;
13
14     function Img (I : Integer) return String
15         renames Integer'Image;
16
17     V : Vector := 20 & 10 & 0 & 13;
18 begin
19     Put_Line ("Vector elements are: ");
20
21     --
22     -- Using for ... of loop to iterate:
23     --
24     for E of V loop
25         Put_Line ("- " & Img (E));
26     end loop;
27
28 end Show_Vector_Iteration;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Iteration
MD5: 4fc9a939aa822097d3a937646d3e2910

Runtime output

```

Vector elements are:
- 20
- 10
- 0
- 13

```

This code displays each element from the vector V.

Because we're given a reference, we can display not only the value of an element but also modify it. For example, we could easily write a loop to add one to each element of vector V:

```

for E of V loop
    E := E + 1;
end loop;

```

We can also use indices to access vector elements. The format is similar to a loop over array elements: we use a **for I in <range>** loop. The range is provided by V.First_Index and V.Last_Index. We can access the current element by using it as an array index: V (I). For example:

Listing 284: show_vector_index_iteration.adb

```

1 with Ada.Containers.Vectors;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Vector_Index_Iteration is
6
7     package Integer_Vectors is new
8         Ada.Containers.Vectors
9         (Index_Type => Natural,
10          Element_Type => Integer);
11
12     use Integer_Vectors;

```

(continues on next page)

(continued from previous page)

```

13
14   V : Vector := 20 & 10 & 0 & 13;
15 begin
16   Put_Line ("Vector elements are: ");
17
18   --
19   -- Using indices in a "for I in ..." loop
20   -- to iterate:
21   --
22   for I in V.First_Index .. V.Last_Index loop
23     -- Displaying current index I
24     Put ("- ["
25         & Extended_Index'Image (I)
26         & "]" );
27
28     Put (Integer'Image (V (I)));
29
30     -- We could also use the V.Element (I)
31     -- function to retrieve the element at
32     -- the current index I
33
34     New_Line;
35   end loop;
36
37 end Show_Vector_Index_Iteration;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Index_Iteration
MD5: f5600bbcc53d6d6887a771b1505676e9

Runtime output

```

Vector elements are:
- [ 0]  20
- [ 1]  10
- [ 2]   0
- [ 3]  13

```

Here, in addition to displaying the vector elements, we're also displaying each index, *I*, just like what we can do for array indices. Also, we can access the element by using either the short form `V (I)` or the longer form `V.Element (I)` but not `V.I`.

As mentioned in the previous section, you can use cursors to iterate over containers. For this, use the function `Iterate`, which retrieves a cursor for each position in the vector. The corresponding loop has the format `for C in V.Iterate loop`. Like the previous example using indices, you can again access the current element by using the cursor as an array index: `V (C)`. For example:

Listing 285: show_vector_cursor_iteration.adb

```

1 with Ada.Containers.Vectors;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Vector_Cursor_Iteration is
6
7   package Integer_Vectors is new
8     Ada.Containers.Vectors
9     (Index_Type => Natural,
10      Element_Type => Integer);

```

(continues on next page)

(continued from previous page)

```

11
12  use Integer_Vectors;
13
14  V : Vector := 20 & 10 & 0 & 13;
15  begin
16    Put_Line ("Vector elements are: ");
17
18    --
19    --  Use a cursor to iterate in a loop:
20    --
21    for C in V.Iterate loop
22      --  Using To_Index function to retrieve
23      --  the index for the cursor position
24      Put ("- ["
25          & Extended_Index'Image (To_Index (C))
26          & "] ");
27
28      Put (Integer'Image (V (C)));
29
30      --  We could use Element (C) to retrieve
31      --  the vector element for the cursor
32      --  position
33
34      New_Line;
35    end loop;
36
37    --  Alternatively, we could iterate with a
38    --  while-loop:
39    --
40    --  declare
41    --    C : Cursor := V.First;
42    --  begin
43    --    while C /= No_Element loop
44    --      some processing here...
45    --
46    --    C := Next (C);
47    --  end loop;
48    --  end;
49
50  end Show_Vector_Cursor_Iteration;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Cursor_Iteration
MD5: de789bbd2e1814aae3fb5213c99ac25c

Runtime output

```

Vector elements are:
- [ 0]  20
- [ 1]  10
- [ 2]   0
- [ 3]  13

```

Instead of accessing an element in the loop using `V (C)`, we could also have used the longer form `Element (C)`. In this example, we're using the function `To_Index` to retrieve the index corresponding to the current cursor.

As shown in the comments after the loop, we could also use a `while ... loop` to iterate over the vector. In this case, we would start with a cursor for the first element (retrieved by calling `V.First`) and then call `Next (C)` to retrieve a cursor for subsequent elements. `Next (C)` returns `No_Element` when the cursor reaches the end of the vector.

You can directly modify the elements using a reference. This is what it looks like when using both indices and cursors:

```
-- Modify vector elements using index
for I in V.First_Index .. V.Last_Index loop
  V (I) := V (I) + 1;
end loop;

-- Modify vector elements using cursor
for C in V.Iterate loop
  V (C) := V (C) + 1;
end loop;
```

The Reference Manual requires that the worst-case complexity for accessing an element be $O(\log N)$.

Another way of modifying elements of a vector is using a *process procedure*, which takes an individual element and does some processing on it. You can call `Update_Element` and pass both a cursor and an access to the process procedure. For example:

Listing 286: show_vector_update.adb

```
1 with Ada.Containers.Vectors;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Vector_Update is
6
7   package Integer_Vectors is new
8     Ada.Containers.Vectors
9     (Index_Type => Natural,
10      Element_Type => Integer);
11
12   use Integer_Vectors;
13
14   procedure Add_One (I : in out Integer) is
15   begin
16     I := I + 1;
17   end Add_One;
18
19   V : Vector := 20 & 10 & 12;
20 begin
21   --
22   -- Use V.Update_Element to process elements
23   --
24   for C in V.Iterate loop
25     V.Update_Element (C, Add_One'Access);
26   end loop;
27
28 end Show_Vector_Update;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Update
MD5: 5dcc3dd8020632a8ea2ce975ecd8f4da

19.1.6 Finding and changing elements

You can locate a specific element in a vector by retrieving its index. `Find_Index` retrieves the index of the first element matching the value you're looking for. Alternatively, you can use `Find` to retrieve a cursor referencing that element. For example:

Listing 287: show_find_vector_element.adb

```

1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Find_Vector_Element is
6
7      package Integer_Vectors is new
8          Ada.Containers.Vectors
9              (Index_Type => Natural,
10               Element_Type => Integer);
11
12      use Integer_Vectors;
13
14      V : Vector := 20 & 10 & 0 & 13;
15      Idx : Extended_Index;
16      C : Cursor;
17  begin
18      -- Using Find_Index to retrieve the index
19      -- of element with value 10
20      Idx := V.Find_Index (10);
21      Put_Line ("Index of element with value 10 is "
22                & Extended_Index'Image (Idx));
23
24      -- Using Find to retrieve the cursor for
25      -- the element with value 13
26      C := V.Find (13);
27      Idx := To_Index (C);
28      Put_Line ("Index of element with value 13 is "
29                & Extended_Index'Image (Idx));
30  end Show_Find_Vector_Element;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Find_Vector_Element
MD5: c3da01cd66c8705a7cbccae8390d5f81

Runtime output

```

Index of element with value 10 is  1
Index of element with value 13 is  3

```

As we saw in the previous section, we can directly access vector elements by using either an index or cursor. However, an exception is raised if we try to access an element with an invalid index or cursor, so we must check whether the index or cursor is valid before using it to access an element. In our example, `Find_Index` or `Find` might not have found the element in the vector. We check for this possibility by comparing the index to `No_Index` or the cursor to `No_Element`. For example:

```

-- Modify vector element using index
if Idx /= No_Index then
    V (Idx) := 11;
end if;

-- Modify vector element using cursor
if C /= No_Element then
    V (C) := 14;
end if;

```

Instead of writing `V (C) := 14`, we could use the longer form `V.Replace_Element (C, 14)`.

19.1.7 Inserting elements

In the previous sections, we've seen examples of how to add elements to a vector:

- using the concatenation operator (&) at the vector declaration, or
- calling the Prepend and Append procedures.

You may want to insert an element at a specific position, e.g. before a certain element in the vector. You do this by calling Insert. For example:

Listing 288: show_vector_insert.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Insert is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      use Integer_Vectors;
14
15      procedure Show_Elements (V : Vector) is
16      begin
17          New_Line;
18          Put_Line ("Vector has "
19                  & Count_Type'Image (V.Length)
20                  & " elements");
21
22          if not V.Is_Empty then
23              Put_Line ("Vector elements are: ");
24              for E of V loop
25                  Put_Line ("- " & Integer'Image (E));
26              end loop;
27          end if;
28      end Show_Elements;
29
30      V : Vector := 20 & 10 & 12;
31      C : Cursor;
32  begin
33      Show_Elements (V);
34
35      New_Line;
36      Put_Line ("Adding element with value 9");
37      Put_Line (" (before 10)...");
38
39      --
40      -- Using V.Insert to insert the element
41      -- into the vector
42      --
43      C := V.Find (10);
44      if C /= No_Element then
45          V.Insert (C, 9);
46      end if;
47
48      Show_Elements (V);
49
50  end Show_Vector_Insert;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Insert
MD5: af49f390388896c51ab97541036fbcaf

Runtime output

```
Vector has 3 elements
Vector elements are:
- 20
- 10
- 12

Adding element with value 9
(before 10)...

Vector has 4 elements
Vector elements are:
- 20
- 9
- 10
- 12
```

In this example, we're looking for an element with the value of 10. If we find it, we insert an element with the value of 9 before it.

19.1.8 Removing elements

You can remove elements from a vector by passing either a valid index or cursor to the Delete procedure. If we combine this with the functions Find_Index and Find from the previous section, we can write a program that searches for a specific element and deletes it, if found:

Listing 289: show_remove_vector_element.adb

```
1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Remove_Vector_Element is
6      package Integer_Vectors is new
7          Ada.Containers.Vectors
8              (Index_Type => Natural,
9               Element_Type => Integer);
10
11     use Integer_Vectors;
12
13     V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
14     Idx : Extended_Index;
15     C : Cursor;
16 begin
17     -- Use Find_Index to retrieve index of
18     -- the element with value 10
19     Idx := V.Find_Index (10);
20
21     -- Checking whether index is valid
22     if Idx /= No_Index then
23         -- Removing element using V.Delete
24         V.Delete (Idx);
25     end if;
```

(continues on next page)

(continued from previous page)

```

26
27  -- Use Find to retrieve cursor for
28  -- the element with value 13
29  C := V.Find (13);
30
31  -- Check whether index is valid
32  if C /= No_Element then
33      -- Remove element using V.Delete
34      V.Delete (C);
35  end if;
36
37 end Show_Remove_Vector_Element;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Remove_Vector_Element
MD5: 540d0dc5715e58926e9dc4600bd6ad5d

We can extend this approach to delete all elements matching a certain value. We just need to keep searching for the element in a loop until we get an invalid index or cursor. For example:

Listing 290: show_remove_vector_elements.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Remove_Vector_Elements is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      use Integer_Vectors;
14
15      procedure Show_Elements (V : Vector) is
16      begin
17          New_Line;
18          Put_Line ("Vector has "
19                  & Count_Type'Image (V.Length)
20                  & " elements");
21
22          if not V.Is_Empty then
23              Put_Line ("Vector elements are: ");
24              for E of V loop
25                  Put_Line ("- " & Integer'Image (E));
26              end loop;
27          end if;
28      end Show_Elements;
29
30      V : Vector := 20 & 10 & 0 & 13 & 10 & 14 & 13;
31  begin
32      Show_Elements (V);
33
34      --
35      -- Remove elements using an index
36      --
37  declare

```

(continues on next page)

(continued from previous page)

```

38     E : constant Integer := 10;
39     I : Extended_Index;
40     begin
41         New_Line;
42         Put_Line
43             ("Removing all elements with value of "
44              & Integer'Image (E) & "...");
45         loop
46             I := V.Find_Index (E);
47             exit when I = No_Index;
48             V.Delete (I);
49         end loop;
50     end;
51
52     --
53     -- Remove elements using a cursor
54     --
55     declare
56         E : constant Integer := 13;
57         C : Cursor;
58     begin
59         New_Line;
60         Put_Line
61             ("Removing all elements with value of "
62              & Integer'Image (E) & "...");
63         loop
64             C := V.Find (E);
65             exit when C = No_Element;
66             V.Delete (C);
67         end loop;
68     end;
69
70     Show_Elements (V);
71 end Show_Remove_Vector_Elements;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Remove_Vector_Elements
MD5: 6e364843b9638224bd9a36eb9d45e446

Runtime output

```

Vector has 7 elements
Vector elements are:
- 20
- 10
- 0
- 13
- 10
- 14
- 13

Removing all elements with value of 10...

Removing all elements with value of 13...

Vector has 3 elements
Vector elements are:
- 20
- 0

```

(continues on next page)

(continued from previous page)

- 14

In this example, we remove all elements with the value 10 from the vector by retrieving their index. Likewise, we remove all elements with the value 13 by retrieving their cursor.

19.1.9 Other Operations

We've seen some operations on vector elements. Here, we'll see operations on the vector as a whole. The most prominent is the concatenation of multiple vectors, but we'll also see operations on vectors, such as sorting and sorted merging operations, that view the vector as a sequence of elements and operate on the vector considering the element's relations to each other.

We do vector concatenation using the `&` operator on vectors. Let's consider two vectors `V1` and `V2`. We can concatenate them by doing `V := V1 & V2`. `V` contains the resulting vector.

The generic package `Generic_Sorting` is a child package of `Ada.Containers.Vectors`. It contains sorting and merging operations. Because it's a generic package, you can't use it directly, but have to instantiate it. In order to use these operations on a vector of integer values (`Integer_Vectors`, in our example), you need to instantiate it directly as a child of `Integer_Vectors`. The next example makes it clear how to do this.

After instantiating `Generic_Sorting`, we make all the operations available to us with the `use` statement. We can then call `Sort` to sort the vector and `Merge` to merge one vector into another.

The following example presents code that manipulates three vectors (`V1`, `V2`, `V3`) using the concatenation, sorting and merging operations:

Listing 291: show_vector_ops.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Ops is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10             (Index_Type => Natural,
11              Element_Type => Integer);
12
13      package Integer_Vectors_Sorting is
14          new Integer_Vectors.Generic_Sorting;
15
16      use Integer_Vectors;
17      use Integer_Vectors_Sorting;
18
19      procedure Show_Elements (V : Vector) is
20          begin
21              New_Line;
22              Put_Line ("Vector has "
23                      & Count_Type'Image (V.Length)
24                      & " elements");
25
26              if not V.Is_Empty then
27                  Put_Line ("Vector elements are: ");
28                  for E of V loop
29                      Put_Line ("- " & Integer'Image (E));
30                  end loop;

```

(continues on next page)

(continued from previous page)

```

31     end if;
32 end Show_Elements;
33
34 V, V1, V2, V3 : Vector;
35 begin
36   V1 := 10 & 12 & 18;
37   V2 := 11 & 13 & 19;
38   V3 := 15 & 19;
39
40   New_Line;
41   Put_Line ("---- V1 ----");
42   Show_Elements (V1);
43
44   New_Line;
45   Put_Line ("---- V2 ----");
46   Show_Elements (V2);
47
48   New_Line;
49   Put_Line ("---- V3 ----");
50   Show_Elements (V3);
51
52   New_Line;
53   Put_Line
54     ("Concatenating V1, V2 and V3 into V:");
55
56   V := V1 & V2 & V3;
57
58   Show_Elements (V);
59
60   New_Line;
61   Put_Line ("Sorting V:");
62
63   Sort (V);
64
65   Show_Elements (V);
66
67   New_Line;
68   Put_Line ("Merging V2 into V1:");
69
70   Merge (V1, V2);
71
72   Show_Elements (V1);
73
74 end Show_Vector_Ops;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Vector_Ops
MD5: 3301513e4e7fd2f28488966e5b24e448

Runtime output

```

---- V1 ----

Vector has 3 elements
Vector elements are:
- 10
- 12
- 18

```

(continues on next page)

(continued from previous page)

```
---- V2 ----  
  
Vector has 3 elements  
Vector elements are:  
- 11  
- 13  
- 19  
  
---- V3 ----  
  
Vector has 2 elements  
Vector elements are:  
- 15  
- 19  
  
Concatenating V1, V2 and V3 into V:  
  
Vector has 8 elements  
Vector elements are:  
- 10  
- 12  
- 18  
- 11  
- 13  
- 19  
- 15  
- 19  
  
Sorting V:  
  
Vector has 8 elements  
Vector elements are:  
- 10  
- 11  
- 12  
- 13  
- 15  
- 18  
- 19  
- 19  
  
Merging V2 into V1:  
  
Vector has 6 elements  
Vector elements are:  
- 10  
- 11  
- 12  
- 13  
- 18  
- 19
```

The Reference Manual requires that the worst-case complexity of a call to Sort be $O(N^2)$ and the average complexity be better than $O(N^2)$.

19.2 Sets

Sets are another class of containers. While vectors allow duplicated elements to be inserted, sets ensure that no duplicated elements exist.

In the following sections, we'll see operations you can perform on sets. However, since many of the operations on vectors are similar to the ones used for sets, we'll cover them more quickly here. Please refer back to the section on vectors for a more detailed discussion.

19.2.1 Initialization and iteration

To initialize a set, you can call the `Insert` procedure. However, if you do, you need to ensure no duplicate elements are being inserted: if you try to insert a duplicate, you'll get an exception. If you have less control over the elements to be inserted so that there may be duplicates, you can use another option instead:

- a version of `Insert` that returns a Boolean value indicating whether the insertion was successful;
- the `Include` procedure, which silently ignores any attempt to insert a duplicated element.

To iterate over a set, you can use a `for E of S` loop, as you saw for vectors. This gives you a reference to each element in the set.

Let's see an example:

Listing 292: `show_set_init.adb`

```
1 with Ada.Containers; use Ada.Containers;
2 with Ada.Containers.Ordered_Sets;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Set_Init is
7
8     package Integer_Sets is new
9         Ada.Containers.Ordered_Sets
10            (Element_Type => Integer);
11
12     use Integer_Sets;
13
14     S : Set;
15     -- Same as: S : Integer_Sets.Set;
16     C : Cursor;
17     Ins : Boolean;
18 begin
19     S.Insert (20);
20     S.Insert (10);
21     S.Insert (0);
22     S.Insert (13);
23
24     -- Calling S.Insert(0) now would raise
25     -- Constraint_Error because this element
26     -- is already in the set. We instead call a
27     -- version of Insert that doesn't raise an
28     -- exception but instead returns a Boolean
29     -- indicating the status
30
31     S.Insert (0, C, Ins);
32     if not Ins then
33         Put_Line
34             ("Error while inserting 0 into set");
35     end if;
36
37     -- We can also call S.Include instead
```

(continues on next page)

(continued from previous page)

```

38  -- If the element is already present,
39  -- the set remains unchanged
40  S.Include (0);
41  S.Include (13);
42  S.Include (14);
43
44  Put_Line ("Set has "
45           & Count_Type'Image (S.Length)
46           & " elements");
47
48  --
49  -- Iterate over set using for .. of loop
50  --
51  Put_Line ("Elements:");
52  for E of S loop
53      Put_Line ("- " & Integer'Image (E));
54  end loop;
55 end Show_Set_Init;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Set_Init
MD5: b87f6729fea278396347248b95a30cb6

Runtime output

```

Error while inserting 0 into set
Set has 5 elements
Elements:
- 0
- 10
- 13
- 14
- 20

```

19.2.2 Operations on elements

In this section, we briefly explore the following operations on sets:

- Delete and Exclude to remove elements;
- Contains and Find to verify the existence of elements.

To delete elements, you call the procedure `Delete`. However, analogously to the `Insert` procedure above, `Delete` raises an exception if the element to be deleted isn't present in the set. If you want to permit the case where an element might not exist, you can call `Exclude`, which silently ignores any attempt to delete a non-existent element.

`Contains` returns a Boolean value indicating whether a value is contained in the set. `Find` also looks for an element in a set, but returns a cursor to the element or `No_Element` if the element doesn't exist. You can use either function to search for elements in a set.

Let's look at an example that makes use of these operations:

Listing 293: show_set_element_ops.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Ordered_Sets;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5

```

(continues on next page)

(continued from previous page)

```

6  procedure Show_Set_Element_Ops is
7
8      package Integer_Sets is new
9          Ada.Containers.Ordered_Sets
10             (Element_Type => Integer);
11
12      use Integer_Sets;
13
14      procedure Show_Elements (S : Set) is
15      begin
16          New_Line;
17          Put_Line ("Set has "
18                  & Count_Type'Image (S.Length)
19                  & " elements");
20          Put_Line ("Elements:");
21          for E of S loop
22              Put_Line ("- " & Integer'Image (E));
23          end loop;
24      end Show_Elements;
25
26      S : Set;
27  begin
28      S.Insert (20);
29      S.Insert (10);
30      S.Insert (0);
31      S.Insert (13);
32
33      S.Delete (13);
34
35      -- Calling S.Delete (13) again raises
36      -- Constraint_Error because the element
37      -- is no longer present in the set, so
38      -- it can't be deleted. We can call
39      -- V.Exclude instead:
40      S.Exclude (13);
41
42      if S.Contains (20) then
43          Put_Line ("Found element 20 in set");
44      end if;
45
46      -- Alternatively, we could use S.Find
47      -- instead of S.Contains
48      if S.Find (0) /= No_Element then
49          Put_Line ("Found element 0 in set");
50      end if;
51
52      Show_Elements (S);
53  end Show_Set_Element_Ops;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Set_Element_Ops
MD5: 77fb2aaba4221e337b0f90dd1a49c556

Runtime output

```

Found element 20 in set
Found element 0 in set

```

```

Set has  3 elements
Elements:

```

(continues on next page)

(continued from previous page)

- 0
 - 10
 - 20

In addition to ordered sets used in the examples above, the standard library also offers hashed sets. The Reference Manual requires the following average complexity of each operation:

Operations	Ordered_Sets	Hashed_Sets
• Insert • Include • Replace • Delete • Exclude • Find	$O((\log N)^2)$ or better	$O(\log N)$
Subprogram using cursor	$O(1)$	$O(1)$

19.2.3 Other Operations

The previous sections mostly dealt with operations on individual elements of a set. But Ada also provides typical set operations: union, intersection, difference and symmetric difference. In contrast to some vector operations we've seen before (e.g. Merge), here you can use built-in operators, such as -. The following table lists the operations and its associated operator:

Set Operation	Operator
Union	or
Intersection	and
Difference	-
Symmetric difference	xor

The following example makes use of these operators:

Listing 294: show_set_ops.adb

```
1 with Ada.Containers; use Ada.Containers;
2 with Ada.Containers.Ordered_Sets;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Set_Ops is
7
8     package Integer_Sets is new
9         Ada.Containers.Ordered_Sets
10             (Element_Type => Integer);
11
12     use Integer_Sets;
13
14     procedure Show_Elements (S : Set) is
15     begin
16         Put_Line ("Elements:");
17         for E of S loop
18             Put_Line ("- " & Integer'Image (E));
```

(continues on next page)

(continued from previous page)

```

19     end loop;
20 end Show_Elements;
21
22 procedure Show_Op (S      : Set;
23                   Op_Name : String) is
24 begin
25     New_Line;
26     Put_Line (Op_Name
27              & "(set #1, set #2) has "
28              & Count_Type'Image (S.Length)
29              & " elements");
30 end Show_Op;
31
32 S1, S2, S3 : Set;
33 begin
34     S1.Insert (0);
35     S1.Insert (10);
36     S1.Insert (13);
37
38     S2.Insert (0);
39     S2.Insert (10);
40     S2.Insert (14);
41
42     S3.Insert (0);
43     S3.Insert (10);
44
45     New_Line;
46     Put_Line ("---- Set #1 ----");
47     Show_Elements (S1);
48
49     New_Line;
50     Put_Line ("---- Set #2 ----");
51     Show_Elements (S2);
52
53     New_Line;
54     Put_Line ("---- Set #3 ----");
55     Show_Elements (S3);
56
57     New_Line;
58     if S3.Is_Subset (S1) then
59         Put_Line ("S3 is a subset of S1");
60     else
61         Put_Line ("S3 is not a subset of S1");
62     end if;
63
64     S3 := S1 and S2;
65     Show_Op (S3, "Intersection");
66     Show_Elements (S3);
67
68     S3 := S1 or S2;
69     Show_Op (S3, "Union");
70     Show_Elements (S3);
71
72     S3 := S1 - S2;
73     Show_Op (S3, "Difference");
74     Show_Elements (S3);
75
76     S3 := S1 xor S2;
77     Show_Op (S3, "Symmetric difference");
78     Show_Elements (S3);
79

```

(continues on next page)

(continued from previous page)

80 `end Show_Set_Ops;`**Code block metadata**

Project: Courses.Intro_To_Ada.Standard_Library.Show_Set_Ops
 MD5: be9086591fc643e53facaf2ffea6c26d

Runtime output

```

---- Set #1 ----
Elements:
- 0
- 10
- 13

---- Set #2 ----
Elements:
- 0
- 10
- 14

---- Set #3 ----
Elements:
- 0
- 10

S3 is a subset of S1

Intersection(set #1, set #2) has 2 elements
Elements:
- 0
- 10

Union(set #1, set #2) has 4 elements
Elements:
- 0
- 10
- 13
- 14

Difference(set #1, set #2) has 1 elements
Elements:
- 13

Symmetric difference(set #1, set #2) has 2 elements
Elements:
- 13
- 14

```

19.3 Indefinite maps

The previous sections presented containers for elements of definite types. Although most examples in those sections presented **Integer** types as element type of the containers, containers can also be used with indefinite types, an example of which is the **String** type. However, indefinite types require a different kind of containers designed specially for them.

We'll also be exploring a different class of containers: maps. They associate a key with a specific value. An example of a map is the one-to-one association between a person and their age. If we consider a person's name to be the key, the value is the person's age.

19.3.1 Hashed maps

Hashed maps are maps that make use of a hash as a key. The hash itself is calculated by a function you provide.

In other languages

Hashed maps are similar to dictionaries in Python and hashes in Perl. One of the main differences is that these scripting languages allow using different types for the values contained in a single map, while in Ada, both the type of key and value are specified in the package instantiation and remains constant for that specific map. You can't have a map where two elements are of different types or two keys are of different types. If you want to use multiple types, you must create a different map for each and use only one type in each map.

When instantiating a hashed map from `Ada.Containers.Indefinite_Hashed_Maps`, we specify following elements:

- `Key_Type`: type of the key
- `Element_Type`: type of the element
- `Hash`: hash function for the `Key_Type`
- `Equivalent_Keys`: an equality operator (e.g. `=`) that indicates whether two keys are to be considered equal.
 - If the type specified in `Key_Type` has a standard operator, you can use it, which you do by specifying that operator as the value of `Equivalent_Keys`.

In the next example, we'll use a string as a key type. We'll use the `Hash` function provided by the standard library for strings (in the `Ada.Strings` package) and the standard equality operator.

You add elements to a hashed map by calling `Insert`. If an element is already contained in a map `M`, you can access it directly by using its key. For example, you can change the value of an element by calling `M ("My_Key") := 10`. If the key is not found, an exception is raised. To verify if a key is available, use the function `Contains` (as we've seen above in the section on sets).

Let's see an example:

Listing 295: `show_hashed_map.adb`

```

1  with Ada.Containers.Indefinite_Hashed_Maps;
2  with Ada.Strings.Hash;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Hashed_Map is
7
8      package Integer_Hashed_Maps is new
9          Ada.Containers.Indefinite_Hashed_Maps
10             (Key_Type      => String,
11              Element_Type  => Integer,
12              Hash          => Ada.Strings.Hash,
13              Equivalent_Keys => "=");
14
15      use Integer_Hashed_Maps;
16
17      M : Map;
18      -- Same as:

```

(continues on next page)

(continued from previous page)

```

19  --
20  -- M : Integer_Hashed_Maps.Map;
21  begin
22    M.Include ("Alice", 24);
23    M.Include ("John", 40);
24    M.Include ("Bob", 28);
25
26    if M.Contains ("Alice") then
27      Put_Line ("Alice's age is "
28        & Integer'Image (M ("Alice")));
29    end if;
30
31    -- Update Alice's age
32    -- Key must already exist in M.
33    -- Otherwise an exception is raised.
34    M ("Alice") := 25;
35
36    New_Line; Put_Line ("Name & Age:");
37    for C in M.Iterate loop
38      Put_Line (Key (C) & ": "
39        & Integer'Image (M (C)));
40    end loop;
41
42  end Show_Hashed_Map;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Hashed_Map
MD5: 6117775bd9ce2b1466f448b100117ded

Runtime output

```

Alice's age is  24

Name & Age:
John:  40
Bob:   28
Alice:  25

```

19.3.2 Ordered maps

Ordered maps share many features with hashed maps. The main differences are:

- A hash function isn't needed. Instead, you must provide an ordering function (< operator), which the ordered map will use to order elements and allow fast access, $O(\log N)$, using a binary search.
 - If the type specified in `Key_Type` has a standard < operator, you can use it in a similar way as we did for `Equivalent_Keys` above for hashed maps.

Let's see an example:

Listing 296: show_ordered_map.adb

```

1  with Ada.Containers.Indefinite_Ordered_Maps;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Ordered_Map is
6
7    package Integer_Ordered_Maps is new

```

(continues on next page)

(continued from previous page)

```

8      Ada.Containers.Indefinite_Ordered_Maps
9      (Key_Type      => String,
10       Element_Type  => Integer);
11
12      use Integer_Ordered_Maps;
13
14      M : Map;
15  begin
16      M.Include ("Alice", 24);
17      M.Include ("John", 40);
18      M.Include ("Bob", 28);
19
20      if M.Contains ("Alice") then
21          Put_Line ("Alice's age is "
22                  & Integer'Image (M ("Alice")));
23      end if;
24
25      -- Update Alice's age
26      -- Key must already exist in M
27      M ("Alice") := 25;
28
29      New_Line; Put_Line ("Name & Age:");
30      for C in M.Iterate loop
31          Put_Line (Key (C) & ": "
32                  & Integer'Image (M (C)));
33      end loop;
34
35  end Show_Ordered_Map;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Ordered_Map
MD5: 3deb3c685e767cee271b06e87727b086

Runtime output

```

Alice's age is  24

Name & Age:
Alice:  25
Bob:    28
John:   40

```

You can see a great similarity between the examples above and from the previous section. In fact, since both kinds of maps share many operations, we didn't need to make extensive modifications when we changed our example to use ordered maps instead of hashed maps. The main difference is seen when we run the examples: the output of a hashed map is usually unordered, but the output of a ordered map is always ordered, as implied by its name.

19.3.3 Complexity

Hashed maps are generally the fastest data structure available to you in Ada if you need to associate heterogeneous keys to values and search for them quickly. In most cases, they are slightly faster than ordered maps. So if you don't need ordering, use hashed maps.

The Reference Manual requires the following average complexity of operations:

Operations	Ordered_Maps	Hashed_Maps
• Insert • Include • Replace • Delete • Exclude • Find	$O((\log N)^2)$ or better	$O(\log N)$
Subprogram using cursor	$O(1)$	$O(1)$

STANDARD LIBRARY: DATES & TIMES

The standard library supports processing of dates and times using two approaches:

- *Calendar* approach, which is suitable for handling dates and times in general;
- *Real-time* approach, which is better suited for real-time applications that require enhanced precision — for example, by having access to an absolute clock and handling time spans. Note that this approach only supports times, not dates.

The following sections present these two approaches.

20.1 Date and time handling

The Ada.Calendar package supports handling of dates and times. Let's look at a simple example:

Listing 297: display_current_time.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Calendar; use Ada.Calendar;
3
4 with Ada.Calendar.Formatting;
5 use Ada.Calendar.Formatting;
6
7 procedure Display_Current_Time is
8     Now : Time := Clock;
9 begin
10     Put_Line ("Current time: " & Image (Now));
11 end Display_Current_Time;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Current_Time
MD5: 4a88069b33ecf80314b0164a472ff606

Runtime output

Current time: 2025-12-27 15:41:31

This example displays the current date and time, which is retrieved by a call to the Clock function. We call the function Image from the Ada.Calendar.Formatting package to get a **String** for the current date and time. We could instead retrieve each component using the Split function. For example:

Listing 298: display_current_year.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Calendar; use Ada.Calendar;
```

(continues on next page)

(continued from previous page)

```

3
4 procedure Display_Current_Year is
5     Now          : Time := Clock;
6
7     Now_Year     : Year_Number;
8     Now_Month    : Month_Number;
9     Now_Day      : Day_Number;
10    Now_Seconds   : Day_Duration;
11 begin
12     Split (Now,
13           Now_Year,
14           Now_Month,
15           Now_Day,
16           Now_Seconds);
17
18     Put_Line ("Current year is: "
19             & Year_Number'Image (Now_Year));
20     Put_Line ("Current month is: "
21             & Month_Number'Image (Now_Month));
22     Put_Line ("Current day is: "
23             & Day_Number'Image (Now_Day));
24 end Display_Current_Year;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Current_Year
MD5: fdf298ee97f225261ce3839ebd833bbe

Runtime output

```

Current year is: 2025
Current month is: 12
Current day is: 27

```

Here, we're retrieving each element and displaying it separately.

20.1.1 Delaying using date

You can delay an application so that it restarts at a specific date and time. We saw something similar in the chapter on tasking. You do this using a **delay until** statement. For example:

Listing 299: display_delay_next_specific_time.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Calendar; use Ada.Calendar;
3
4 with Ada.Calendar.Formatting;
5 use Ada.Calendar.Formatting;
6
7 with Ada.Calendar.Time_Zones;
8 use Ada.Calendar.Time_Zones;
9
10 procedure Display_Delay_Next_Specific_Time is
11     TZ : Time_Offset := UTC_Time_Offset;
12     Next : Time :=
13         Ada.Calendar.Formatting.Time_Of
14         (Year      => 2018,
15          Month     => 5,
16          Day       => 1,

```

(continues on next page)

(continued from previous page)

```

17      Hour      => 15,
18      Minute    => 0,
19      Second     => 0,
20      Sub_Second => 0.0,
21      Leap_Second => False,
22      Time_Zone  => TZ);
23
24      -- Next = 2018-05-01 15:00:00.00
25      --      (local time-zone)
26  begin
27      Put_Line ("Let's wait until...");
28      Put_Line (Image (Next, True, TZ));
29
30      delay until Next;
31
32      Put_Line ("Enough waiting!");
33  end Display_Delay_Next_Specific_Time;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Delay_Next_Specific_Time
MD5: 36ec2bdce7c1e8d107fae54ef9852d3f

Runtime output

```

Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!

```

In this example, we specify the date and time by initializing `Next` using a call to `Time_Of`, a function taking the various components of a date (year, month, etc) and returning an element of the `Time` type. Because the date specified is in the past, the **delay until** statement won't produce any noticeable effect. However, if we passed a date in the future, the program would wait until that specific date and time arrived.

Here we're converting the time to the local timezone. If we don't specify a timezone, *Coordinated Universal Time* (abbreviated to UTC) is used by default. By retrieving the time offset to UTC with a call to `UTC_Time_Offset` from the `Ada.Calendar.Time_Zones` package, we can initialize `TZ` and use it in the call to `Time_Of`. This is all we need to do to make the information provided to `Time_Of` relative to the local time zone.

We could achieve a similar result by initializing `Next` with a **String**. We can do this with a call to `Value` from the `Ada.Calendar.Formatting` package. This is the modified code:

Listing 300: `display_delay_next_specific_time.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Calendar; use Ada.Calendar;
3
4  with Ada.Calendar.Formatting;
5  use Ada.Calendar.Formatting;
6
7  with Ada.Calendar.Time_Zones;
8  use Ada.Calendar.Time_Zones;
9
10 procedure Display_Delay_Next_Specific_Time is
11     TZ : Time_Offset := UTC_Time_Offset;
12     Next : Time :=
13         Ada.Calendar.Formatting.Value
14         ("2018-05-01 15:00:00.00", TZ);
15

```

(continues on next page)

(continued from previous page)

```

16  -- Next = 2018-05-01 15:00:00.00
17  --      (local time-zone)
18  begin
19    Put_Line ("Let's wait until...");
20    Put_Line (Image (Next, True, TZ));
21
22    delay until Next;
23
24    Put_Line ("Enough waiting!");
25  end Display_Delay_Next_Specific_Time;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Delay_Next_Specific_Time
MD5: fdf6ad7fca303d4d7bd444c23e11c7bd

Runtime output

```

Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!

```

In this example, we're again using TZ in the call to Value to adjust the input time to the current time zone.

In the examples above, we were delaying to a specific date and time. Just like we saw in the tasking chapter, we could instead specify the delay relative to the current time. For example, we could delay by 5 seconds, using the current time:

Listing 301: display_delay_next.adb

```

1  with Ada.Calendar; use Ada.Calendar;
2  with Ada.Text_IO;  use Ada.Text_IO;
3
4  procedure Display_Delay_Next is
5    D : Duration := 5.0;
6    --      ^ seconds
7    Now : Time    := Clock;
8    Next : Time   := Now + D;
9    --      ^ use duration to
10   --      specify next
11   --      point in time
12  begin
13    Put_Line ("Let's wait "
14              & Duration'Image (D)
15              & " seconds...");
16    delay until Next;
17    Put_Line ("Enough waiting!");
18  end Display_Delay_Next;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Delay_Next
MD5: 58360d93388c3fe027c3d9d67389efc7

Runtime output

```

Let's wait 5.000000000 seconds...
Enough waiting!

```

Here, we're specifying a duration of 5 seconds in D, adding it to the current time from Now, and storing the sum in Next. We then use it in the **delay until** statement.

20.2 Real-time

In addition to `Ada.Calendar`, the standard library also supports time operations for real-time applications. These are included in the `Ada.Real_Time` package. This package also includes a `Time` type. However, in the `Ada.Real_Time` package, the `Time` type is used to represent an absolute clock and handle a time span. This contrasts with the `Ada.Calendar`, which uses the `Time` type to represent dates and times.

In the previous section, we used the `Time` type from the `Ada.Calendar` and the `delay until` statement to delay an application by 5 seconds. We could have used the `Ada.Real_Time` package instead. Let's modify that example:

Listing 302: `display_delay_next_real_time.adb`

```

1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Ada.Real_Time; use Ada.Real_Time;
3
4 procedure Display_Delay_Next_Real_Time is
5     D      : Time_Span := Seconds (5);
6     Next   : Time      := Clock + D;
7 begin
8     Put_Line ("Let's wait "
9               & Duration'Image (To_Duration (D))
10              & " seconds...");
11     delay until Next;
12     Put_Line ("Enough waiting!");
13 end Display_Delay_Next_Real_Time;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Standard_Library.Display_Delay_Next_Real_Time`
MD5: `a80e96c4ac7bd3ba7813f983b10cb038`

Runtime output

```

Let's wait  5.0000000000 seconds...
Enough waiting!
```

The main difference is that `D` is now a variable of type `Time_Span`, defined in the `Ada.Real_Time` package. We call the function `Seconds` to initialize `D`, but could have gotten a finer granularity by calling `Nanoseconds` instead. Also, we need to first convert `D` to the `Duration` type using `To_Duration` before we can display it.

20.2.1 Benchmarking

One interesting application using the `Ada.Real_Time` package is benchmarking. We've used that package before in a previous section when discussing tasking. Let's look at an example of benchmarking:

Listing 303: `display_benchmarking.adb`

```

1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Ada.Real_Time; use Ada.Real_Time;
3
4 procedure Display_Benchmarking is
5
6     procedure Computational_Intensive_App is
7     begin
8         delay 5.0;
9     end Computational_Intensive_App;
```

(continues on next page)

(continued from previous page)

```

10
11   Start_Time, Stop_Time : Time;
12   Elapsed_Time         : Time_Span;
13
14 begin
15   Start_Time := Clock;
16
17   Computational_Intensive_App;
18
19   Stop_Time := Clock;
20   Elapsed_Time := Stop_Time - Start_Time;
21
22   Put_Line ("Elapsed time: "
23           & Duration'Image
24           (To_Duration (Elapsed_Time))
25           & " seconds");
26 end Display_Benchmarking;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Benchmarking
MD5: 4b20940cb613d3f634be5224f409efeb

Runtime output

Elapsed time: 5.002554479 seconds

This example defines a dummy `Computational_Intensive_App` implemented using a simple `delay` statement. We initialize `Start_Time` and `Stop_Time` from the then-current clock and calculate the elapsed time. By running this program, we see that the time is roughly 5 seconds, which is expected due to the `delay` statement.

A similar application is benchmarking of CPU time. We can implement this using the `Execution_Time` package. Let's modify the previous example to measure CPU time:

Listing 304: `display_benchmarking_cpu_time.adb`

```

1 with Ada.Text_IO;          use Ada.Text_IO;
2 with Ada.Real_Time;        use Ada.Real_Time;
3 with Ada.Execution_Time;   use Ada.Execution_Time;
4
5 procedure Display_Benchmarking_CPU_Time is
6
7   procedure Computational_Intensive_App is
8   begin
9     delay 5.0;
10  end Computational_Intensive_App;
11
12  Start_Time, Stop_Time : CPU_Time;
13  Elapsed_Time         : Time_Span;
14
15 begin
16   Start_Time := Clock;
17
18   Computational_Intensive_App;
19
20   Stop_Time := Clock;
21   Elapsed_Time := Stop_Time - Start_Time;
22
23   Put_Line ("CPU time: "
24           & Duration'Image

```

(continues on next page)

(continued from previous page)

```

25         (To_Duration (Elapsed_Time))
26         & " seconds");
27 end Display_Benchmarking_CPU_Time;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Benchmarking_CPU_Time
MD5: ba83ddbd05db523479be5692c4134901

Runtime output

CPU time: 0.000050028 seconds

In this example, `Start_Time` and `Stop_Time` are of type `CPU_Time` instead of `Time`. However, we still call the `Clock` function to initialize both variables and calculate the elapsed time in the same way as before. By running this program, we see that the CPU time is significantly lower than the 5 seconds we've seen before. This is because the **delay** statement doesn't require much CPU time. The results will be different if we change the implementation of `Computational_Intensive_App` to use a mathematical function in a long loop. For example:

Listing 305: display_benchmarking_math.adb

```

1  with Ada.Text_IO;          use Ada.Text_IO;
2  with Ada.Real_Time;        use Ada.Real_Time;
3  with Ada.Execution_Time;   use Ada.Execution_Time;
4
5  with Ada.Numerics.Generic_Elementary_Functions;
6
7  procedure Display_Benchmarking_Math is
8
9      procedure Computational_Intensive_App is
10         package Funcs is new
11             Ada.Numerics.Generic_Elementary_Functions
12             (Float_Type => Long_Long_Float);
13         use Funcs;
14
15         X : Long_Long_Float;
16     begin
17         for I in 0 .. 1_000_000 loop
18             X := Tan (Arctan
19                 (Tan (Arctan
20                     (Tan (Arctan
21                         (Tan (Arctan
22                             (Tan (Arctan
23                                 (Tan (Arctan
24                                     (0.577)))))))))))));
25         end loop;
26     end Computational_Intensive_App;
27
28     procedure Benchm_Elapsed_Time is
29         Start_Time, Stop_Time : Time;
30         Elapsed_Time          : Time_Span;
31
32     begin
33         Start_Time := Clock;
34
35         Computational_Intensive_App;
36
37         Stop_Time := Clock;
38         Elapsed_Time := Stop_Time - Start_Time;

```

(continues on next page)

(continued from previous page)

```
39
40     Put_Line ("Elapsed time: "
41               & Duration'Image
42               (To_Duration (Elapsed_Time))
43               & " seconds");
44 end Benchm_Elapsed_Time;
45
46 procedure Benchm_CPU_Time is
47     Start_Time, Stop_Time : CPU_Time;
48     Elapsed_Time          : Time_Span;
49
50 begin
51     Start_Time := Clock;
52
53     Computational_Intensive_App;
54
55     Stop_Time := Clock;
56     Elapsed_Time := Stop_Time - Start_Time;
57
58     Put_Line ("CPU time: "
59               & Duration'Image
60               (To_Duration (Elapsed_Time))
61               & " seconds");
62 end Benchm_CPU_Time;
63 begin
64     Benchm_Elapsed_Time;
65     Benchm_CPU_Time;
66 end Display_Benchmarking_Math;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Display_Benchmarking_Math
MD5: 06fe96bf03321c248ddled843648cf0b

Runtime output

Elapsed time: 0.574971306 seconds
CPU time: 0.574500344 seconds

Now that our dummy `Computational_Intensive_App` involves mathematical operations requiring significant CPU time, the measured elapsed and CPU time are much closer to each other than before.

STANDARD LIBRARY: STRINGS

In previous chapters, we've seen source-code examples using the **String** type, which is a fixed-length string type — essentially, it's an array of characters. In many cases, this data type is good enough to deal with textual information. However, there are situations that require more advanced text processing. Ada offers alternative approaches for these cases:

- *Bounded strings*: similar to fixed-length strings, bounded strings have a maximum length, which is set at its instantiation. However, bounded strings are not arrays of characters. At any time, they can contain a string of varied length — provided this length is below or equal to the maximum length.
- *Unbounded strings*: similar to bounded strings, unbounded strings can contain strings of varied length. However, in addition to that, they don't require a maximum length to be specified at the declaration of a string. In this sense, they are very flexible.

For further reading...

Although we don't specify a maximum length for unbounded strings, the limit is defined by the Reference Manual²⁰:

An object of type `Unbounded_String` represents a **String** whose low bound is 1 and whose length can vary conceptually between 0 and **Natural'Last**.

Therefore, the implicit maximum length is **Natural'Last**. In contrast, bounded strings have an explicit maximum length that is specified when the **Generic_Bounded_Length** package is instantiated (as we'll see *later on* (page 246)).

Another difference between bounded and unbounded strings is the strategy that is used by the compiler to allocate memory for those strings. When using GNAT, bounded strings are allocated on the stack, while unbounded strings are allocated on the heap.

The following sections present an overview of the different string types and common operations for string types.

21.1 String operations

Operations on standard (fixed-length) strings are available in the `Ada.Strings.Fixed` package. As mentioned previously, standard strings are arrays of elements of **Character** type with a *fixed-length*. That's why this child package is called `Fixed`.

One of the simplest operations provided is counting the number of substrings available in a string (**Count**) and finding their corresponding indices (**Index**). Let's look at an example:

²⁰ <http://www.ada-auth.org/standards/12rm/html/RM-A-4-5.html>

Listing 306: show_find_substring.adb

```

1 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Show_Find_Substring is
5
6     S : String := "Hello" & 3 * " World";
7     P : constant String := "World";
8     Idx : Natural;
9     Cnt : Natural;
10
11 begin
12     Cnt := Ada.Strings.Fixed.Count
13         (Source => S,
14          Pattern => P);
15
16     Put_Line ("String: " & S);
17     Put_Line ("Count for '" & P & "': "
18               & Natural'Image (Cnt));
19
20     Idx := 0;
21     for I in 1 .. Cnt loop
22         Idx := Index
23             (Source => S,
24              Pattern => P,
25              From   => Idx + 1);
26
27         Put_Line ("Found instance of '"
28                 & P & "' at position: "
29                 & Natural'Image (Idx));
30     end loop;
31 end Show_Find_Substring;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Find_Substring
MD5: faa8373bf9aec9f9f5507cf55590b0c0

Runtime output

```

String: Hello World World World
Count for 'World': 3
Found instance of 'World' at position: 7
Found instance of 'World' at position: 13
Found instance of 'World' at position: 19

```

We initialize the string *S* using a multiplication. Writing `"Hello" & 3 * " World"` creates the string `Hello World World World`. We then call the function `Count` to get the number of instances of the word `World` in *S*. Next we call the function `Index` in a loop to find the index of each instance of `World` in *S*.

That example looked for instances of a specific substring. In the next example, we retrieve all the words in the string. We do this using `Find-Token` and specifying whitespaces as separators. For example:

Listing 307: show_find_words.adb

```

1 with Ada.Strings;      use Ada.Strings;
2 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3 with Ada.Strings.Maps; use Ada.Strings.Maps;
4 with Ada.Text_IO;      use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```

5
6 procedure Show_Find_Words is
7
8   S   : String := "Hello" & 3 * " World";
9   F   : Positive;
10  L   : Natural;
11  I   : Natural := 1;
12
13  Whitespace : constant Character_Set :=
14    To_Set (' ');
15 begin
16   Put_Line ("String: " & S);
17   Put_Line ("String length: "
18     & Integer'Image (S'Length));
19
20   while I in S'Range loop
21     Find-Token
22       (Source => S,
23        Set    => Whitespace,
24        From   => I,
25        Test   => Outside,
26        First  => F,
27        Last   => L);
28
29     exit when L = 0;
30
31     Put_Line ("Found word instance at position "
32       & Natural'Image (F)
33       & ": '" & S (F .. L) & "'");
34     -- & "-" & F'Img & "-" & L'Img
35
36     I := L + 1;
37   end loop;
38 end Show_Find_Words;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Find_Words
MD5: e622f489af5901e5d31f314efc3324d2

Runtime output

```

String: Hello World World World
String length: 23
Found word instance at position 1: 'Hello'
Found word instance at position 7: 'World'
Found word instance at position 13: 'World'
Found word instance at position 19: 'World'

```

We pass a set of characters to be used as delimiters to the procedure `Find-Token`. This set is a member of the `Character_Set` type from the `Ada.Strings.Maps` package. We call the `To_Set` function (from the same package) to initialize the set to `Whitespace` and then call `Find-Token` to loop over each valid index and find the starting index of each word. We pass `Outside` to the `Test` parameter of the `Find-Token` procedure to indicate that we're looking for indices that are outside the `Whitespace` set, i.e. actual words. The `First` and `Last` parameters of `Find-Token` are output parameters that indicate the valid range of the substring. We use this information to display the string `S (F .. L)`.

The operations we've looked at so far read strings, but don't modify them. We next discuss operations that change the content of strings:

Operation	Description
Insert	Insert substring in a string
Overwrite	Overwrite a string with a substring
Delete	Delete a substring
Trim	Remove whitespaces from a string

All these operations are available both as functions or procedures. Functions create a new string but procedures perform the operations in place. The procedure will raise an exception if the constraints of the string are not satisfied. For example, if we have a string *S* containing 10 characters, inserting a string with two characters (e.g. "!!") into it produces a string containing 12 characters. Since it has a fixed length, we can't increase its size. One possible solution in this case is to specify that truncation should be applied while inserting the substring. This keeps the length of *S* fixed. Let's see an example that makes use of both function and procedure versions of Insert, Overwrite, and Delete:

Listing 308: show_adapted_strings.adb

```

1  with Ada.Strings;           use Ada.Strings;
2  with Ada.Strings.Fixed;     use Ada.Strings.Fixed;
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  procedure Show_Adapted_Strings is
6
7      S : String := "Hello World";
8      P : constant String := "World";
9      N : constant String := "Beautiful";
10
11     procedure Display_Adapted_String
12         (Source : String;
13          Before : Positive;
14          New_Item : String;
15          Pattern : String)
16     is
17         S_Ins_In : String := Source;
18         S_Ovr_In : String := Source;
19         S_Del_In : String := Source;
20
21         S_Ins : String :=
22             Insert (Source,
23                   Before,
24                   New_Item & " ");
25         S_Ovr : String :=
26             Overwrite (Source,
27                      Before,
28                      New_Item);
29         S_Del : String :=
30             Trim (Delete (Source,
31                          Before,
32                          Before +
33                              Pattern'Length - 1),
34                  Ada.Strings.Right);
35     begin
36         Insert (S_Ins_In,
37               Before,
38               New_Item,
39               Right);
40
41         Overwrite (S_Ovr_In,
42                   Before,

```

(continues on next page)

(continued from previous page)

```

43         New_Item,
44         Right);
45
46     Delete (S_Del_In,
47            Before,
48            Before + Pattern'Length - 1);
49
50     Put_Line ("Original:  '"
51              & Source & "'");
52
53     Put_Line ("Insert:    '"
54              & S_Ins & "'");
55     Put_Line ("Overwrite: '"
56              & S_Ovr & "'");
57     Put_Line ("Delete:    '"
58              & S_Del & "'");
59
60     Put_Line ("Insert      (in-place): '"
61              & S_Ins_In & "'");
62     Put_Line ("Overwrite (in-place): '"
63              & S_Ovr_In & "'");
64     Put_Line ("Delete      (in-place): '"
65              & S_Del_In & "'");
66 end Display_Adapted_String;
67
68 Idx : Natural;
69 begin
70     Idx := Index
71         (Source => S,
72          Pattern => P);
73
74     if Idx > 0 then
75         Display_Adapted_String (S, Idx, N, P);
76     end if;
77 end Show_Adapted_Strings;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Adapted_Strings
MD5: b31b6bc94d8bdbc717c6b6b2534beb6

Runtime output

```

Original:  'Hello World'
Insert:    'Hello Beautiful World'
Overwrite: 'Hello Beautiful'
Delete:    'Hello'
Insert      (in-place): 'Hello Beaut'
Overwrite (in-place): 'Hello Beaut'
Delete      (in-place): 'Hello      '

```

In this example, we look for the index of the substring World and perform operations on this substring within the outer string. The procedure `Display_Adapted_String` uses both versions of the operations. For the procedural version of `Insert` and `Overwrite`, we apply truncation to the right side of the string (`Right`). For the `Delete` procedure, we specify the range of the substring, which is replaced by whitespaces. For the function version of `Delete`, we also call `Trim` which trims the trailing whitespace.

21.2 Limitation of fixed-length strings

Using fixed-length strings is usually good enough for strings that are initialized when they are declared. However, as seen in the previous section, procedural operations on strings cause difficulties when done on fixed-length strings because fixed-length strings are arrays of characters. The following example shows how cumbersome the initialization of fixed-length strings can be when it's not performed in the declaration:

Listing 309: show_char_array.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2
3  procedure Show_Char_Array is
4      S : String (1 .. 15);
5      -- Strings are arrays of Character
6  begin
7      S := "Hello           ";
8      -- Alternatively:
9      --
10     -- #1:
11     --     S (1 .. 5)      := "Hello";
12     --     S (6 .. S'Last) := (others => ' ');
13     --
14     -- #2:
15     --     S := ('H', 'e', 'l', 'l', 'o',
16     --           others => ' ');
17
18     Put_Line ("String: " & S);
19     Put_Line ("String Length: "
20               & Integer'Image (S'Length));
21 end Show_Char_Array;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Char_Array
MD5: 9f3df03c9c5336184139cf2a22f2cb7e

Runtime output

```
String: Hello
String Length:  15
```

In this case, we can't simply write `S := "Hello"` because the resulting array of characters for the `Hello` constant has a different length than the `S` string. Therefore, we need to include trailing whitespaces to match the length of `S`. As shown in the example, we could use an exact range for the initialization (`S (1 .. 5)`) or use an explicit array of individual characters.

When strings are initialized or manipulated at run-time, it's usually better to use bounded or unbounded strings. An important feature of these types is that they aren't arrays, so the difficulties presented above don't apply. Let's start with bounded strings.

21.3 Bounded strings

Bounded strings are defined in the `Ada.Strings.Bounded.Generic_Bounded_Length` package. Because this is a generic package, you need to instantiate it and set the maximum length of the bounded string. You can then declare bounded strings of the `Bounded_String` type.

Both bounded and fixed-length strings have a maximum length that they can hold. How-

ever, bounded strings are not arrays, so initializing them at run-time is much easier. For example:

Listing 310: show_bounded_string.adb

```

1  with Ada.Strings;           use Ada.Strings;
2  with Ada.Strings.Bounded;
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  procedure Show_Bounded_String is
6      package B_Str is new
7          Ada.Strings.Bounded.Generic_Bounded_Length
8              (Max => 15);
9      use B_Str;
10
11     S1, S2 : Bounded_String;
12
13     procedure Display_String_Info
14         (S : Bounded_String)
15     is
16     begin
17         Put_Line ("String: " & To_String (S));
18         Put_Line ("String Length: "
19             & Integer'Image (Length (S)));
20         -- String:
21         --     S'Length => ok
22         -- Bounded_String:
23         --     S'Length => compilation error:
24         --         bounded strings are
25         --         not arrays!
26
27         Put_Line ("Max. Length: "
28             & Integer'Image (Max_Length));
29     end Display_String_Info;
30
31 begin
32     S1 := To_Bounded_String ("Hello");
33     Display_String_Info (S1);
34
35     S2 := To_Bounded_String ("Hello World");
36     Display_String_Info (S2);
37
38     S1 := To_Bounded_String
39         ("Something longer to say here...",
40         Right);
41     Display_String_Info (S1);
42 end Show_Bounded_String;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Bounded_String
MD5: a51fdeacfd43923145ee92bf5c72ecd6

Runtime output

```

String: Hello
String Length: 5
Max. Length: 15
String: Hello World
String Length: 11
Max. Length: 15
String: Something longe

```

(continues on next page)

(continued from previous page)

```
String Length: 15
Max. Length: 15
```

By using bounded strings, we can easily assign to S1 and S2 multiple times during execution. We use the `To_Bounded_String` and `To_String` functions to convert, in the respective direction, between fixed-length and bounded strings. A call to `To_Bounded_String` raises an exception if the length of the input string is greater than the maximum capacity of the bounded string. To avoid this, we can use the truncation parameter (`Right` in our example).

Bounded strings are not arrays, so we can't use the `'Length` attribute as we did for fixed-length strings. Instead, we call the `Length` function, which returns the length of the bounded string. The `Max_Length` constant represents the maximum length of the bounded string that we set when we instantiated the package.

After initializing a bounded string, we can manipulate it. For example, we can append a string to a bounded string using `Append` or concatenate bounded strings using the `&` operator. Like so:

Listing 311: show_bounded_string_op.adb

```

1  with Ada.Strings;           use Ada.Strings;
2  with Ada.Strings.Bounded;
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  procedure Show_Bounded_String_Op is
6      package B_Str is new
7          Ada.Strings.Bounded.Generic_Bounded_Length
8              (Max => 30);
9      use B_Str;
10
11     S1, S2 : Bounded_String;
12  begin
13     S1 := To_Bounded_String ("Hello");
14     -- Alternatively:
15     --
16     -- A := Null_Bounded_String & "Hello";
17
18     Append (S1, " World");
19     -- Alternatively:
20     -- Append (A, " World", Right);
21
22     Put_Line ("String: " & To_String (S1));
23
24     S2 := To_Bounded_String ("Hello!");
25     S1 := S1 & " " & S2;
26     Put_Line ("String: " & To_String (S1));
27  end Show_Bounded_String_Op;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Standard_Library.Show_Bounded_String_Op
MD5: c7c6a840c314a9cd9f75aac082a63159
```

Runtime output

```
String: Hello World
String: Hello World Hello!
```

We can initialize a bounded string with an empty string using the `Null_Bounded_String` constant. Also, we can use the `Append` procedure and specify the truncation mode like we do with the `To_Bounded_String` function.

21.4 Unbounded strings

Unbounded strings are defined in the `Ada.Strings.Unbounded` package. This is *not* a generic package, so we don't need to instantiate it before using the `Unbounded_String` type. As you may recall from the previous section, bounded strings require a package instantiation.

Unbounded strings are similar to bounded strings. The main difference is that they can hold strings of any size and adjust according to the input string: if we assign, e.g., a 10-character string to an unbounded string and later assign a 50-character string, internal operations in the container ensure that memory is allocated to store the new string. In most cases, developers don't need to worry about these operations. Also, no truncation is necessary.

Initialization of unbounded strings is very similar to bounded strings. Let's look at an example:

Listing 312: `show_unbounded_string.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Strings; use Ada.Strings;
3
4 with Ada.Strings.Unbounded;
5 use Ada.Strings.Unbounded;
6
7 procedure Show_Unbounded_String is
8   S1, S2 : Unbounded_String;
9
10  procedure Display_String_Info
11    (S : Unbounded_String)
12  is
13  begin
14    Put_Line ("String: " & To_String (S));
15    Put_Line ("String Length: "
16              & Integer'Image (Length (S)));
17  end Display_String_Info;
18 begin
19   S1 := To_Unbounded_String ("Hello");
20   -- Alternatively:
21   --
22   -- A := Null_Unbounded_String & "Hello";
23
24   Display_String_Info (S1);
25
26   S2 := To_Unbounded_String ("Hello World");
27   Display_String_Info (S2);
28
29   S1 := To_Unbounded_String
30         ("Something longer to say here...");
31   Display_String_Info (S1);
32 end Show_Unbounded_String;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Standard_Library.Show_Unbounded_String`
MD5: `904402992c96eb393b875d1b7cf49c1b`

Runtime output

```
String: Hello
String Length: 5
String: Hello World
```

(continues on next page)

(continued from previous page)

```
String Length: 11
String: Something longer to say here...
String Length: 31
```

Like bounded strings, we can assign to `S1` and `S2` multiple times during execution and use the `To_Unbounded_String` and `To_String` functions to convert back-and-forth between fixed-length strings and unbounded strings. However, in this case, truncation is not needed.

And, just like for bounded strings, you can use the `Append` procedure and the `&` operator for unbounded strings. For example:

Listing 313: `show_unbounded_string_op.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.Unbounded;
4 use Ada.Strings.Unbounded;
5
6 procedure Show_Unbounded_String_Op is
7   S1, S2 : Unbounded_String :=
8           Null_Unbounded_String;
9 begin
10   S1 := S1 & "Hello";
11   S2 := S2 & "Hello!";
12
13   Append (S1, " World");
14   Put_Line ("String: " & To_String (S1));
15
16   S1 := S1 & " " & S2;
17   Put_Line ("String: " & To_String (S1));
18 end Show_Unbounded_String_Op;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Standard_Library.Show_Unbounded_String_Op
MD5: 806e24a6dd0bc87e76f73a22e42ba390
```

Runtime output

```
String: Hello World
String: Hello World Hello!
```

In this example, we're concatenating the unbounded `S1` and `S2` strings with the `"Hello"` and `"Hello!"` strings, respectively. Also, we're using the `Append` procedure, just like we did with bounded strings.

STANDARD LIBRARY: FILES AND STREAMS

Ada provides different approaches for file input/output (I/O):

- *Text I/O*, which supports file I/O in text format, including the display of information on the console.
- *Sequential I/O*, which supports file I/O in binary format written in a sequential fashion for a specific data type.
- *Direct I/O*, which supports file I/O in binary format for a specific data type, but also supporting access to any position of a file.
- *Stream I/O*, which supports I/O of information for multiple data types, including objects of unbounded types, using files in binary format.

This table presents a summary of the features we've just seen:

File I/O option	Format	Random access	Data types
Text I/O	text		string type
Sequential I/O	binary		single type
Direct I/O	binary	✓	single type
Stream I/O	binary	✓	multiple types

In the following sections, we discuss details about these I/O approaches.

22.1 Text I/O

In most parts of this course, we used the `Put_Line` procedure to display information on the console. However, this procedure also accepts a **File_Type** parameter. For example, you can select between standard output and standard error by setting this parameter explicitly:

Listing 314: `show_std_text_out.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Std_Text_Out is
4 begin
5   Put_Line (Standard_Output, "Hello World #1");
6   Put_Line (Standard_Error, "Hello World #2");
7 end Show_Std_Text_Out;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Std_Text_Out
MD5: 4d75bd2906226897244e3d2a611c9725

Runtime output

```
Hello World #1
Hello World #2
```

You can also use this parameter to write information to any text file. To create a new file for writing, use the `Create` procedure, which initializes a **File_Type** element that you can later pass to `Put_Line` (instead of, e.g., `Standard_Output`). After you finish writing information, you can close the file by calling the `Close` procedure.

You use a similar method to read information from a text file. However, when opening the file, you must specify that it's an input file (`In_File`) instead of an output file. Also, instead of calling the `Put_Line` procedure, you call the `Get_Line` function to read information from the file.

Let's see an example that writes information into a new text file and then reads it back from the same file:

Listing 315: `show_simple_text_file_io.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Text_File_IO is
4     F      : File_Type;
5     File_Name : constant String := "simple.txt";
6 begin
7     Create (F, Out_File, File_Name);
8     Put_Line (F, "Hello World #1");
9     Put_Line (F, "Hello World #2");
10    Put_Line (F, "Hello World #3");
11    Close (F);
12
13    Open (F, In_File, File_Name);
14    while not End_Of_File (F) loop
15        Put_Line (Get_Line (F));
16    end loop;
17    Close (F);
18 end Show_Simple_Text_File_IO;
```

Code block metadata

```
Project: Courses.Intro_To_Ada.Standard_Library.Show_Simple_Text_File_IO
MD5: 7461e946eef18c93219fa4ce3afbblea
```

Runtime output

```
Hello World #1
Hello World #2
Hello World #3
```

In addition to the `Create` and `Close` procedures, the standard library also includes a `Reset` procedure, which, as the name implies, resets (erases) all the information from the file. For example:

Listing 316: `show_text_file_reset.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Text_File_Reset is
4     F      : File_Type;
5     File_Name : constant String := "simple.txt";
6 begin
7     Create (F, Out_File, File_Name);
8     Put_Line (F, "Hello World #1");
```

(continues on next page)

(continued from previous page)

```

9   Reset (F);
10  Put_Line (F, "Hello World #2");
11  Close (F);
12
13  Open (F, In_File, File_Name);
14  while not End_Of_File (F) loop
15      Put_Line (Get_Line (F));
16  end loop;
17  Close (F);
18  end Show_Text_File_Reset;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Text_File_Reset
MD5: 5e5498f03b2c829513af062c5959fc93

Runtime output

Hello World #2

By running this program, we notice that, although we've written the first string ("Hello World #1") to the file, it has been erased because of the call to Reset.

In addition to opening a file for reading or writing, you can also open an existing file and append to it. Do this by calling the Open procedure with the Append_File option.

When calling the Open procedure, an exception is raised if the specified file isn't found. Therefore, you should handle exceptions in that context. The following example deletes a file and then tries to open the same file for reading:

Listing 317: show_text_file_input_except.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Text_File_Input_Except is
4      F      : File_Type;
5      File_Name : constant String := "simple.txt";
6  begin
7      -- Open output file and delete it
8      Create (F, Out_File, File_Name);
9      Delete (F);
10
11     -- Try to open deleted file
12     Open (F, In_File, File_Name);
13     Close (F);
14  exception
15     when Name_Error =>
16         Put_Line ("File does not exist");
17     when others =>
18         Put_Line
19             ("Error while processing input file");
20  end Show_Text_File_Input_Except;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Text_File_Input_Except
MD5: c8d257091831c48d10b6e70e34b4261b

Runtime output

File does not exist

In this example, we create the file by calling `Create` and then delete it by calling `Delete`. After the call to `Delete`, we can no longer use the **File_Type** element. After deleting the file, we try to open the non-existent file, which raises a `Name_Error` exception.

22.2 Sequential I/O

The previous section presented details about text file I/O. Here, we discuss doing file I/O in binary format. The first package we'll explore is the `Ada.Sequential_IO` package. Because this package is a generic package, you need to instantiate it for the data type you want to use for file I/O. Once you've done that, you can use the same procedures we've seen in the previous section: `Create`, `Open`, `Close`, `Reset` and `Delete`. However, instead of calling the `Get_Line` and `Put_Line` procedures, you'd call the `Read` and `Write` procedures.

In the following example, we instantiate the `Ada.Sequential_IO` package for floating-point types:

Listing 318: `show_seq_float_io.adb`

```
1 with Ada.Text_IO;
2 with Ada.Sequential_IO;
3
4 procedure Show_Seq_Float_IO is
5   package Float_IO is
6     new Ada.Sequential_IO (Float);
7   use Float_IO;
8
9   F      : Float_IO.File_Type;
10  File_Name : constant String :=
11             "float_file.bin";
12 begin
13   Create (F, Out_File, File_Name);
14   Write (F, 1.5);
15   Write (F, 2.4);
16   Write (F, 6.7);
17   Close (F);
18
19   declare
20     Value : Float;
21   begin
22     Open (F, In_File, File_Name);
23     while not End_Of_File (F) loop
24       Read (F, Value);
25       Ada.Text_IO.Put_Line
26         (Float'Image (Value));
27     end loop;
28     Close (F);
29   end;
30 end Show_Seq_Float_IO;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Standard_Library.Show_Seq_Float_IO`
MD5: `27aa5daf92cba5df23fdc55c3578aa34`

Runtime output

```
1.50000E+00
2.40000E+00
6.70000E+00
```

We use the same approach to read and write complex information. The following example

uses a record that includes a Boolean and a floating-point value:

Listing 319: show_seq_rec_io.adb

```

1  with Ada.Text_IO;
2  with Ada.Sequential_IO;
3
4  procedure Show_Seq_Rec_IO is
5      type Num_Info is record
6          Valid : Boolean := False;
7          Value : Float;
8      end record;
9
10     procedure Put_Line (N : Num_Info) is
11     begin
12         if N.Valid then
13             Ada.Text_IO.Put_Line
14                 ("(ok, "
15                  & Float'Image (N.Value) & ")");
16         else
17             Ada.Text_IO.Put_Line
18                 ("(not ok, -----)");
19         end if;
20     end Put_Line;
21
22     package Num_Info_IO is new
23         Ada.Sequential_IO (Num_Info);
24     use Num_Info_IO;
25
26     F      : Num_Info_IO.File_Type;
27     File_Name : constant String :=
28         "float_file.bin";
29 begin
30     Create (F, Out_File, File_Name);
31     Write (F, (True, 1.5));
32     Write (F, (False, 2.4));
33     Write (F, (True, 6.7));
34     Close (F);
35
36     declare
37         Value : Num_Info;
38     begin
39         Open (F, In_File, File_Name);
40         while not End_Of_File (F) loop
41             Read (F, Value);
42             Put_Line (Value);
43         end loop;
44         Close (F);
45     end;
46 end Show_Seq_Rec_IO;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Seq_Rec_IO
MD5: a88b1428cc50745dce0509087e74adb7

Runtime output

```

(ok,      1.50000E+00)
(not ok,  -----)
(ok,      6.70000E+00)
```

As the example shows, we can use the same approach we used for floating-point types to

perform file I/O for this record. Once we instantiate the `Ada.Sequential_IO` package for the record type, file I/O operations are performed the same way.

22.3 Direct I/O

Direct I/O is available in the `Ada.Direct_IO` package. This mechanism is similar to the sequential I/O approach just presented, but allows us to access any position in the file. The package instantiation and most operations are very similar to sequential I/O. To rewrite the `Show_Seq_Float_IO` application presented in the previous section to use the `Ada.Direct_IO` package, we just need to replace the instances of the `Ada.Sequential_IO` package by the `Ada.Direct_IO` package. This is the new source code:

Listing 320: `show_dir_float_io.adb`

```
1 with Ada.Text_IO;
2 with Ada.Direct_IO;
3
4 procedure Show_Dir_Float_IO is
5     package Float_IO is new Ada.Direct_IO (Float);
6     use Float_IO;
7
8     F      : Float_IO.File_Type;
9     File_Name : constant String :=
10         "float_file.bin";
11 begin
12     Create (F, Out_File, File_Name);
13     Write (F, 1.5);
14     Write (F, 2.4);
15     Write (F, 6.7);
16     Close (F);
17
18     declare
19         Value : Float;
20     begin
21         Open (F, In_File, File_Name);
22         while not End_Of_File (F) loop
23             Read (F, Value);
24             Ada.Text_IO.Put_Line
25                 (Float'Image (Value));
26         end loop;
27         Close (F);
28     end;
29 end Show_Dir_Float_IO;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Dir_Float_IO
MD5: e4e5855976de44f53a821eb90dcbb206

Runtime output

```
1.50000E+00
2.40000E+00
6.70000E+00
```

Unlike sequential I/O, direct I/O allows you to access any position in the file. However, it doesn't offer an option to append information to a file. Instead, it provides an `Inout_File` mode allowing reading and writing to a file via the same **File_Type** element.

To access any position in the file, call the `Set_Index` procedure to set the new position / index. You can use the `Index` function to retrieve the current index. Let's see an example:

Listing 321: show_dir_float_in_out_file.adb

```

1  with Ada.Text_IO;
2  with Ada.Direct_IO;
3
4  procedure Show_Dir_Float_In_Out_File is
5      package Float_IO is new Ada.Direct_IO (Float);
6      use Float_IO;
7
8      F          : Float_IO.File_Type;
9      File_Name : constant String :=
10                  "float_file.bin";
11  begin
12      -- Open file for input / output
13      Create (F, Inout_File, File_Name);
14      Write (F, 1.5);
15      Write (F, 2.4);
16      Write (F, 6.7);
17
18      -- Set index to previous position
19      -- and overwrite value
20      Set_Index (F, Index (F) - 1);
21      Write (F, 7.7);
22
23      declare
24          Value : Float;
25      begin
26          -- Set index to start of file
27          Set_Index (F, 1);
28
29          while not End_Of_File (F) loop
30              Read (F, Value);
31              Ada.Text_IO.Put_Line
32                  (Float'Image (Value));
33          end loop;
34          Close (F);
35      end;
36  end Show_Dir_Float_In_Out_File;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Dir_Float_In_Out_File
MD5: 17b83a16ab8fa30f07cf8a0bd54078a1

Runtime output

```

1.50000E+00
2.40000E+00
7.70000E+00

```

By running this example, we see that the file contains 7.7, rather than the previous 6.7 that we wrote. We overwrote the value by changing the index to the previous position before doing another write.

In this example we used the Inout_File mode. Using that mode, we just changed the index back to the initial position before reading from the file (`Set_Index (F, 1)`) instead of closing the file and reopening it for reading.

22.4 Stream I/O

All the previous approaches for file I/O in binary format (sequential and direct I/O) are specific for a single data type (the one we instantiate them with). You can use these approaches to write objects of a single data type that may be an array or record (potentially with many fields), but if you need to create and process files that include different data types, or any objects of an unbounded type, these approaches are not sufficient. Instead, you should use stream I/O.

Stream I/O shares some similarities with the previous approaches. We still use the Create, Open and Close procedures. However, instead of accessing the file directly via a **File_Type** element, you use a **Stream_Access** element. To read and write information, you use the **'Read** or **'Write** attributes of the data types you're reading or writing.

Let's look at a version of the `Show_Dir_Float_IO` procedure from the previous section that makes use of stream I/O instead of direct I/O:

Listing 322: `show_float_stream.adb`

```

1  with Ada.Text_IO;
2
3  with Ada.Streams.Stream_IO;
4  use  Ada.Streams.Stream_IO;
5
6  procedure Show_Float_Stream is
7      F      : File_Type;
8      S      : Stream_Access;
9      File_Name : constant String :=
10                  "float_file.bin";
11  begin
12      Create (F, Out_File, File_Name);
13      S := Stream (F);
14
15      Float'Write (S, 1.5);
16      Float'Write (S, 2.4);
17      Float'Write (S, 6.7);
18
19      Close (F);
20
21      declare
22          Value : Float;
23      begin
24          Open (F, In_File, File_Name);
25          S := Stream (F);
26
27          while not End_Of_File (F) loop
28              Float'Read (S, Value);
29              Ada.Text_IO.Put_Line
30                  (Float'Image (Value));
31          end loop;
32          Close (F);
33      end;
34  end Show_Float_Stream;
```

Code block metadata

Project: `Courses.Intro_To_Ada.Standard_Library.Show_Float_Stream`
MD5: `34ccf04b0821074a332019ac0e38bb3e`

Runtime output

```
1.50000E+00
2.40000E+00
6.70000E+00
```

After the call to `Create`, we retrieve the corresponding `Stream_Access` element by calling the `Stream` function. We then use this stream to write information to the file via the `'Write` attribute of the `Float` type. After closing the file and reopening it for reading, we again retrieve the corresponding `Stream_Access` element and processed to read information from the file via the `'Read` attribute of the `Float` type.

You can use streams to create and process files containing different data types within the same file. You can also read and write unbounded data types such as strings. However, when using unbounded data types you must call the `'Input` and `'Output` attributes of the unbounded data type: these attributes write information about bounds or discriminants in addition to the object's actual data.

The following example shows file I/O that mixes both strings of different lengths and floating-point values:

Listing 323: `show_string_stream.adb`

```
1 with Ada.Text_IO;
2
3 with Ada.Streams.Stream_IO;
4 use  Ada.Streams.Stream_IO;
5
6 procedure Show_String_Stream is
7     F      : File_Type;
8     S      : Stream_Access;
9     File_Name : constant String :=
10         "float_file.bin";
11
12     procedure Output (S : Stream_Access;
13                     FV : Float;
14                     SV : String) is
15     begin
16         String'Output (S, SV);
17         Float'Output (S, FV);
18     end Output;
19
20     procedure Input_Display (S : Stream_Access) is
21         SV : String := String'Input (S);
22         FV : Float  := Float'Input (S);
23     begin
24         Ada.Text_IO.Put_Line (Float'Image (FV)
25                               & " --- " & SV);
26     end Input_Display;
27
28 begin
29     Create (F, Out_File, File_Name);
30     S := Stream (F);
31
32     Output (S, 1.5, "Hi!!");
33     Output (S, 2.4, "Hello world!");
34     Output (S, 6.7, "Something longer here...");
35
36     Close (F);
37
38     Open (F, In_File, File_Name);
39     S := Stream (F);
40
41     while not End_Of_File (F) loop
```

(continues on next page)

(continued from previous page)

```
42     Input_Display (S);
43   end loop;
44   Close (F);
45
46 end Show_String_Stream;
```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_String_Stream
MD5: 3ae8276ada5f24cab49994e368e0fa34

Runtime output

```
1.50000E+00 --- Hi!!
2.40000E+00 --- Hello world!
6.70000E+00 --- Something longer here...
```

When you use Stream I/O, no information is written into the file indicating the type of the data that you wrote. If a file contains data from different types, you must reference types in the same order when reading a file as when you wrote it. If not, the information you get will be corrupted. Unfortunately, strong data typing doesn't help you in this case. Writing simple procedures for file I/O (as in the example above) may help ensuring that the file format is consistent.

Like direct I/O, stream I/O support also allows you to access any location in the file. However, when doing so, you need to be extremely careful that the position of the new index is consistent with the data types you're expecting.

STANDARD LIBRARY: NUMERICS

The standard library provides support for common numeric operations on floating-point types as well as on complex types and matrices. In the sections below, we present a brief introduction to these numeric operations.

23.1 Elementary Functions

The `Ada.Numerics.Elementary_Functions` package provides common operations for floating-point types, such as square root, logarithm, and the trigonometric functions (e.g., `sin`, `cos`). For example:

Listing 324: `show_elem_math.adb`

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Numerics; use Ada.Numerics;
3
4  with Ada.Numerics.Elementary_Functions;
5  use Ada.Numerics.Elementary_Functions;
6
7  procedure Show_Elem_Math is
8      X : Float;
9  begin
10     X := 2.0;
11     Put_Line ("Square root of "
12              & Float'Image (X)
13              & " is "
14              & Float'Image (Sqrt (X)));
15
16     X := e;
17     Put_Line ("Natural log of "
18              & Float'Image (X)
19              & " is "
20              & Float'Image (Log (X)));
21
22     X := 10.0 ** 6.0;
23     Put_Line ("Log_10 of "
24              & Float'Image (X)
25              & " is "
26              & Float'Image (Log (X, 10.0)));
27
28     X := 2.0 ** 8.0;
29     Put_Line ("Log_2 of "
30              & Float'Image (X)
31              & " is "
32              & Float'Image (Log (X, 2.0)));
33
34     X := Pi;
```

(continues on next page)

(continued from previous page)

```

35   Put_Line ("Cos          of "
36             & Float'Image (X)
37             & " is "
38             & Float'Image (Cos (X)));
39
40   X := -1.0;
41   Put_Line ("Arccos       of "
42             & Float'Image (X)
43             & " is "
44             & Float'Image (Arccos (X)));
45 end Show_Elem_Math;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Elem_Math
MD5: 17511d7e17cd98d4b6e49ad302d6dcb6

Runtime output

```

Square root of 2.00000E+00 is 1.41421E+00
Natural log of 2.71828E+00 is 1.00000E+00
Log_10      of 1.00000E+06 is 6.00000E+00
Log_2       of 2.56000E+02 is 8.00000E+00
Cos         of 3.14159E+00 is -1.00000E+00
Arccos      of -1.00000E+00 is 3.14159E+00

```

Here we use the standard `e` and `Pi` constants from the `Ada.Numerics` package.

The `Ada.Numerics.Elementary_Functions` package provides operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. For example, the `Ada.Numerics.Long_Elementary_Functions` package offers the same set of operations for the **Long_Float** type. In addition, the `Ada.Numerics.Generic_Elementary_Functions` package is a generic version of the package that you can instantiate for custom floating-point types. In fact, the `Elementary_Functions` package can be defined as follows:

```

package Elementary_Functions is new
Ada.Numerics.Generic_Elementary_Functions (Float);

```

23.2 Random Number Generation

The `Ada.Numerics.Float_Random` package provides a simple random number generator for the range between 0.0 and 1.0. To use it, declare a generator `G`, which you pass to `Random`. For example:

Listing 325: `show_float_random_num.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Float_Random;
4  use  Ada.Numerics.Float_Random;
5
6  procedure Show_Float_Random_Num is
7      G : Generator;
8      X : Uniformly_Distributed;
9  begin
10     Reset (G);
11

```

(continues on next page)

(continued from previous page)

```

12 Put_Line ("Some random numbers between "
13           & Float'Image
14           (Uniformly_Distributed'First)
15           & " and "
16           & Float'Image
17           (Uniformly_Distributed'Last)
18           & ":");
19 for I in 1 .. 15 loop
20   X := Random (G);
21   Put_Line (Float'Image (X));
22 end loop;
23 end Show_Float_Random_Num;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Float_Random_Num
MD5: cf38ab00e27bad4309010e678113dd36

Runtime output

```

Some random numbers between 0.00000E+00 and 1.00000E+00:
9.80034E-01
2.85511E-01
6.52899E-01
2.38608E-02
2.95568E-02
9.90054E-01
8.09696E-01
1.17431E-02
3.45056E-01
5.46639E-01
8.32065E-01
6.83566E-01
4.59634E-01
7.73473E-01
7.50087E-01

```

The standard library also includes a random number generator for discrete numbers, which is part of the `Ada.Numerics.Discrete_Random` package. Since it's a generic package, you have to instantiate it for the desired discrete type. This allows you to specify a range for the generator. In the following example, we create an application that displays random integers between 1 and 10:

Listing 326: show_discrete_random_num.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics.Discrete_Random;
3
4 procedure Show_Discrete_Random_Num is
5
6   subtype Random_Range is Integer range 1 .. 10;
7
8   package R is new
9     Ada.Numerics.Discrete_Random (Random_Range);
10  use R;
11
12  G : Generator;
13  X : Random_Range;
14 begin
15   Reset (G);
16

```

(continues on next page)

(continued from previous page)

```

17 Put_Line ("Some random numbers between "
18           & Integer'Image (Random_Range'First)
19           & " and "
20           & Integer'Image (Random_Range'Last)
21           & ":");
22
23 for I in 1 .. 15 loop
24     X := Random (G);
25     Put_Line (Integer'Image (X));
26 end loop;
27 end Show_Discrete_Random_Num;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Discrete_Random_Num
MD5: 892f6525477f9a2c56f88885de011fba

Runtime output

```

Some random numbers between 1 and 10:
4
4
10
8
8
2
2
2
4
5
7
2
1
8
5

```

Here, package R is instantiated with the `Random_Range` type, which has a constrained range between 1 and 10. This allows us to control the range used for the random numbers. We could easily modify the application to display random integers between 0 and 20 by changing the specification of the `Random_Range` type. We can also use floating-point or fixed-point types.

23.3 Complex Types

The `Ada.Numerics.Complex_Types` package provides support for complex number types and the `Ada.Numerics.Complex_Elementary_Functions` package provides support for common operations on complex number types, similar to the `Ada.Numerics.Elementary_Functions` package. Finally, you can use the `Ada.Text_IO.Complex_IO` package to perform I/O operations on complex numbers. In the following example, we declare variables of the `Complex` type and initialize them using an aggregate:

Listing 327: show_elem_math.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics; use Ada.Numerics;
3
4 with Ada.Numerics.Complex_Types;
5 use Ada.Numerics.Complex_Types;

```

(continues on next page)

(continued from previous page)

```

6
7 with Ada.Numerics.Complex_Elementary_Functions;
8 use Ada.Numerics.Complex_Elementary_Functions;
9
10 with Ada.Text_IO.Complex_IO;
11
12 procedure Show_Elem_Math is
13
14     package C_IO is new
15         Ada.Text_IO.Complex_IO (Complex_Types);
16     use C_IO;
17
18     X, Y : Complex;
19     R, Th : Float;
20 begin
21     X := (2.0, -1.0);
22     Y := (3.0, 4.0);
23
24     Put (X);
25     Put (" * ");
26     Put (Y);
27     Put (" is ");
28     Put (X * Y);
29     New_Line;
30     New_Line;
31
32     R := 3.0;
33     Th := Pi / 2.0;
34     X := Compose_From_Polar (R, Th);
35     -- Alternatively:
36     -- X := R * Exp ((0.0, Th));
37     -- X := R * e ** Complex'(0.0, Th);
38
39     Put ("Polar form: "
40         & Float'Image (R) & " * e**(i * "
41         & Float'Image (Th) & ")");
42     New_Line;
43
44     Put ("Modulus of ");
45     Put (X);
46     Put (" is ");
47     Put (Float'Image (abs X));
48     New_Line;
49
50     Put ("Argument of ");
51     Put (X);
52     Put (" is ");
53     Put (Float'Image (Argument (X)));
54     New_Line;
55     New_Line;
56
57     Put ("Sqrt of ");
58     Put (X);
59     Put (" is ");
60     Put (Sqrt (X));
61     New_Line;
62 end Show_Elem_Math;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Elem_Math

(continues on next page)

(continued from previous page)

MD5: 73dfc0021c804fa1c5727e272546f172

Runtime output

```
( 2.00000E+00, -1.00000E+00) * ( 3.00000E+00, 4.00000E+00) is ( 1.00000E+01, 5.
↪00000E+00)

Polar form:      3.00000E+00 * e**(i * 1.57080E+00)
Modulus      of (-1.31134E-07, 3.00000E+00) is 3.00000E+00
Argument     of (-1.31134E-07, 3.00000E+00) is 1.57080E+00

Sqrt         of (-1.31134E-07, 3.00000E+00) is ( 1.22474E+00, 1.22474E+00)
```

As we can see from this example, all the common operators, such as `*` and `+`, are available for complex types. You also have typical operations on complex numbers, such as `Argument` and `Exp`. In addition to initializing complex numbers in the cartesian form using aggregates, you can do so from the polar form by calling the `Compose_From_Polar` function.

The `Ada.Numerics.Complex_Types` and `Ada.Numerics.Complex_Elementary_Functions` packages provide operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. In addition, the `Ada.Numerics.Generic_Complex_Types` and `Ada.Numerics.Generic_Complex_Elementary_Functions` packages are generic versions that you can instantiate for custom or pre-defined floating-point types. For example:

```
with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Text_IO.Complex_IO;

procedure Show_Elem_Math is

  package Complex_Types is new
    Ada.Numerics.Generic_Complex_Types (Float);
  use Complex_Types;

  package Elementary_Functions is new
    Ada.Numerics.Generic_Complex_Elementary_Functions
      (Complex_Types);
  use Elementary_Functions;

  package C_IO is new Ada.Text_IO.Complex_IO
    (Complex_Types);
  use C_IO;

  X, Y : Complex;
  R, Th : Float;
```

23.4 Vector and Matrix Manipulation

The `Ada.Numerics.Real_Arrays` package provides support for vectors and matrices. It includes common matrix operations such as inverse, determinant, eigenvalues in addition to simpler operators such as matrix addition and multiplication. You can declare vectors and matrices using the `Real_Vector` and `Real_Matrix` types, respectively.

The following example uses some of the operations from the `Ada.Numerics.Real_Arrays` package:

Listing 328: show_matrix.adb

```

1  with Ada.Text_IO;  use Ada.Text_IO;
2
3  with Ada.Numerics.Real_Arrays;
4  use  Ada.Numerics.Real_Arrays;
5
6  procedure Show_Matrix is
7
8      procedure Put_Vector (V : Real_Vector) is
9      begin
10         Put ("    ");
11         for I in V'Range loop
12             Put (Float'Image (V (I)) & " ");
13         end loop;
14         Put_Line ("");
15     end Put_Vector;
16
17     procedure Put_Matrix (M : Real_Matrix) is
18     begin
19         for I in M'Range (1) loop
20             Put ("    ");
21             for J in M'Range (2) loop
22                 Put (Float'Image (M (I, J)) & " ");
23             end loop;
24             Put_Line ("");
25         end loop;
26     end Put_Matrix;
27
28     V1      : Real_Vector := (1.0, 3.0);
29     V2      : Real_Vector := (75.0, 11.0);
30
31     M1      : Real_Matrix :=
32         ((1.0, 5.0, 1.0),
33          (2.0, 2.0, 1.0));
34     M2      : Real_Matrix :=
35         ((31.0, 11.0, 10.0),
36          (34.0, 16.0, 11.0),
37          (32.0, 12.0, 10.0),
38          (31.0, 13.0, 10.0));
39     M3      : Real_Matrix := ((1.0, 2.0),
40                               (2.0, 3.0));
41 begin
42     Put_Line ("V1");
43     Put_Vector (V1);
44     Put_Line ("V2");
45     Put_Vector (V2);
46     Put_Line ("V1 * V2 =");
47     Put_Line ("    "
48               & Float'Image (V1 * V2));
49     Put_Line ("V1 * V2 =");
50     Put_Matrix (V1 * V2);
51     New_Line;
52
53     Put_Line ("M1");
54     Put_Matrix (M1);
55     Put_Line ("M2");
56     Put_Matrix (M2);
57     Put_Line ("M2 * Transpose(M1) =");
58     Put_Matrix (M2 * Transpose (M1));
59     New_Line;
60

```

(continues on next page)

(continued from previous page)

```

61   Put_Line ("M3");
62   Put_Matrix (M3);
63   Put_Line ("Inverse (M3) =");
64   Put_Matrix (Inverse (M3));
65   Put_Line ("abs Inverse (M3) =");
66   Put_Matrix (abs Inverse (M3));
67   Put_Line ("Determinant (M3) =");
68   Put_Line ("      "
69             & Float'Image (Determinant (M3)));
70   Put_Line ("Solve (M3, V1) =");
71   Put_Vector (Solve (M3, V1));
72   Put_Line ("Eigenvalues (M3) =");
73   Put_Vector (Eigenvalues (M3));
74   New_Line;
75 end Show_Matrix;

```

Code block metadata

Project: Courses.Intro_To_Ada.Standard_Library.Show_Matrix
MD5: c9df45a742a42bd47e03fbf2d0282238

Runtime output

```

V1
  ( 1.00000E+00  3.00000E+00 )
V2
  ( 7.50000E+01  1.10000E+01 )
V1 * V2 =
  1.08000E+02
V1 * V2 =
  ( 7.50000E+01  1.10000E+01 )
  ( 2.25000E+02  3.30000E+01 )

M1
  ( 1.00000E+00  5.00000E+00  1.00000E+00 )
  ( 2.00000E+00  2.00000E+00  1.00000E+00 )
M2
  ( 3.10000E+01  1.10000E+01  1.00000E+01 )
  ( 3.40000E+01  1.60000E+01  1.10000E+01 )
  ( 3.20000E+01  1.20000E+01  1.00000E+01 )
  ( 3.10000E+01  1.30000E+01  1.00000E+01 )
M2 * Transpose(M1) =
  ( 9.60000E+01  9.40000E+01 )
  ( 1.25000E+02  1.11000E+02 )
  ( 1.02000E+02  9.80000E+01 )
  ( 1.06000E+02  9.80000E+01 )

M3
  ( 1.00000E+00  2.00000E+00 )
  ( 2.00000E+00  3.00000E+00 )
Inverse (M3) =
  (-3.00000E+00  2.00000E+00 )
  ( 2.00000E+00 -1.00000E+00 )
abs Inverse (M3) =
  ( 3.00000E+00  2.00000E+00 )
  ( 2.00000E+00  1.00000E+00 )
Determinant (M3) =
  -1.00000E+00
Solve (M3, V1) =
  ( 3.00000E+00 -1.00000E+00 )
Eigenvalues (M3) =

```

(continues on next page)

(continued from previous page)

```
( 4.23607E+00 -2.36068E-01 )
```

Matrix dimensions are automatically determined from the aggregate used for initialization when you don't specify them. You can, however, also use explicit ranges. For example:

```
M1      : Real_Matrix (1 .. 2, 1 .. 3) :=  
          ((1.0, 5.0, 1.0),  
           (2.0, 2.0, 1.0));
```

The `Ada.Numerics.Real_Arrays` package implements operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. In addition, the `Ada.Numerics.Generic_Real_Arrays` package is a generic version that you can instantiate with custom floating-point types. For example, the `Real_Arrays` package can be defined as follows:

```
package Real_Arrays is new  
  Ada.Numerics.Generic_Real_Arrays (Float);
```


APPENDICES

24.1 Appendix A: Generic Formal Types

The following tables contain examples of available formal types for generics:

Formal type	Actual type
Incomplete type Format: <code>type T;</code>	Any type
Discrete type Format: <code>type T is (< >);</code>	Any integer, modular or enumeration type
Range type Format: <code>type T is range < >;</code>	Any signed integer type
Modular type Format: <code>type T is mod < >;</code>	Any modular type
Floating-point type Format: <code>type T is digits < >;</code>	Any floating-point type
Binary fixed-point type Format: <code>type T is delta < >;</code>	Any binary fixed-point type
Decimal fixed-point type Format: <code>type T is delta < > digits < >;</code>	Any decimal fixed-point type
Definite nonlimited private type Format: <code>type T is private;</code>	Any nonlimited, definite type
Nonlimited Private type with discriminant Format: <code>type T (D : DT) is private;</code>	Any nonlimited type with discriminant
Access type Format: <code>type A is access T;</code>	Any access type for type T
Definite derived type Format: <code>type T is new B;</code>	Any concrete type derived from base type B
Limited private type Format: <code>type T is limited private;</code>	Any definite type, limited or not
Incomplete tagged type Format: <code>type T is tagged;</code>	Any concrete, definite, tagged type
Definite tagged private type Format: <code>type T is tagged private;</code>	Any concrete, definite, tagged type
Definite tagged limited private type Format: <code>type T is tagged limited private;</code>	Any concrete definite tagged type, limited or not
Definite abstract tagged private type Format: <code>type T is abstract tagged private;</code>	Any nonlimited, definite tagged type, abstract or concrete

continues on next page

Table 1 – continued from previous page

Formal type	Actual type
Definite abstract tagged limited private type Format: type T is abstract tagged limited private;	Any definite tagged type, limited or not, abstract or concrete
Definite derived tagged type Format: type T is new B with private;	Any concrete tagged type derived from base type B
Definite abstract derived tagged type Format: type T is abstract new B with private;	Any tagged type derived from base type B abstract or concrete
Array type Format: type A is array (R) of T;	Any array type with range R containing elements of type T
Interface type Format: type T is interface;	Any interface type T
Limited interface type Format: type T is limited interface;	Any limited interface type T
Task interface type Format: type T is task interface;	Any task interface type T
Synchronized interface type Format: type T is synchronized interface;	Any synchronized interface type T
Protected interface type Format: type T is protected interface;	Any protected interface type T
Derived interface type Format: type T is new B and I with private;	Any type T derived from base type B and interface I
Derived type with multiple interfaces Format: type T is new B and I1 and I2 with private;	Any type T derived from base type B and interfaces I1 and I2
Abstract derived interface type Format: type T is abstract new B and I with private;	Any type T derived from abstract base type B and interface I
Limited derived interface type Format: type T is limited new B and I with private;	Any type T derived from limited base type B and limited interface I
Abstract limited derived interface type Format: type T is abstract limited new B and I with private;	Any type T derived from abstract limited base type B and limited interface I
Synchronized interface type Format: type T is synchronized new SI with private;	Any type T derived from synchronized interface SI
Abstract synchronized interface type Format: type T is abstract synchronized new SI with private;	Any type T derived from synchronized interface SI

24.1.1 Indefinite version

Many of the examples above can be used for formal indefinite types:

Formal type	Actual type
Indefinite incomplete type Format: <code>type T (<>);</code>	Any type
Indefinite nonlimited private type Format: <code>type T (<>) is private;</code>	Any nonlimited type indefinite or definite
Indefinite limited private type Format: <code>type T (<>) is limited private;</code>	Any type, limited or not, indefinite or definite
Incomplete indefinite tagged private type Format: <code>type T (<>) is tagged;</code>	Any concrete tagged type, indefinite or definite
Indefinite tagged private type Format: <code>type T (<>) is tagged private;</code>	Any concrete, nonlimited tagged type, indefinite or definite
Indefinite tagged limited private type Format: <code>type T (<>) is tagged limited private;</code>	Any concrete tagged type, limited or not, indefinite or definite
Indefinite abstract tagged private type Format: <code>type T (<>) is abstract tagged private;</code>	Any nonlimited tagged type, indefinite or definite, abstract or concrete
Indefinite abstract tagged limited private type Format: <code>type T (<>) is abstract tagged limited private;</code>	Any tagged type, limited or not, indefinite or definite abstract or concrete
Indefinite derived tagged type Format: <code>type T (<>) is new B with private;</code>	Any tagged type derived from base type B, indefinite or definite
Indefinite abstract derived tagged type Format: <code>type T (<>) is abstract new B with private;</code>	Any tagged type derived from base type B, indefinite or definite abstract or concrete

The same examples could also contain discriminants. In this case, (<>) is replaced by a list of discriminants, e.g.: (D: DT).

24.2 Appendix B: Containers

The following table shows all containers available in Ada, including their versions (standard, bounded, unbounded, indefinite):

Category	Container	Std	Bounded	Un-bounded	Indefinite
Vector	Vectors	Y	Y		Y
List	Doubly Linked Lists	Y	Y		Y
Map	Hashed Maps	Y	Y		Y
Map	Ordered Maps	Y	Y		Y
Set	Hashed Sets	Y	Y		Y
Set	Ordered Sets	Y	Y		Y
Tree	Multiway Trees	Y	Y		Y
Generic	Holders				Y
Queue	Synchronized Queue Interfaces	Y			
Queue	Synchronized Queues		Y	Y	
Queue	Priority Queues		Y	Y	

Note

To get the correct container name, replace the whitespace by _ in the names above. (For example, Hashed Maps becomes Hashed_Maps.)

The following table presents the prefixing applied to the container name that depends on its version. As indicated in the table, the standard version does not have a prefix associated with it.

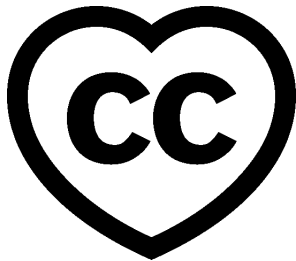
Version	Naming prefix
Std	
Bounded	Bounded_
Unbounded	Unbounded_
Indefinite	Indefinite_

Part II

Advanced Journey With Ada: A Flight In Progress

Copyright © 2019 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)²¹



⚠ Warning

This is work in progress!

Information in this document is subject to change at any time without prior notification.

Note

The code examples in this course use a 50-column limit, which greatly improves the readability of the code on devices with a small screen size. This constraint, however, leads to an unusual coding style. For instance, instead of calling `Put_Line` in a single line, we have this:

```
Put_Line
(" is in the northeast quadrant");
```

or this:

```
Put_Line (" (X => "
& Integer'Image (P.X)
& ")");
```

Note that typical Ada code uses a limit of at least 79 columns. Therefore, please don't take the coding style from this course as a reference!

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](#)²². The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: `Courses.Intro_To_Ada.Imperative_Language.Greet`
- MD5: `cba89a34b87c9dfa71533d982d05e6ab`

you will find it in this directory:

²¹ [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)


```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

This course will teach you advanced topics of the Ada programming language. The *Introduction to Ada* (page 5) course is a prerequisite for this course.

This document was written by Gustavo A. Hoffmann, with major contributions from Robert A. Duff. The document also includes contributions from Franco Gasperoni, Gary Dismukes, Patrick Rogers, and Robert Dewar.

These contributions are clearly indicated in the document, together with the original publication source.

Special thanks to Patrick Rogers for all comments and suggestions. In particular, thanks for sharing the training slides on access types: many ideas from those slides were integrated into this course.

This document was reviewed by Patrick Rogers and Tucker Taft.

CHANGELOG

Changes are being tracked on the CHANGELOG page.

²² https://learn.adacore.com/zip/learning-ada_code.zip

DATA TYPES

25.1 Types

25.1.1 Names

In simple terms, a "name" can be an identifier, i.e. the *name* that we use to refer to an object or a subprogram, for example. This is what we call a *direct name*. However, in Ada, a name can also refer to other language constructs, as we discuss later on in this section.

In the Ada Reference Manual

- 4.1 Name²³

Direct names

Direct names are the simplest form of names in Ada. They can be either identifiers or operator symbols.

Identifiers

An identifier — as the term implies — is a (direct) name that we use to *identify* an object, a subprogram, a type, and so on. When specifying an identifier, we aren't limited to ASCII²⁴ characters: we can use a subset of the Unicode²⁵ standard.

For further reading...

To be more precise, the Normalization Form KC of the Unicode standard is applied to identifiers. You can find more information about it in the Unicode Standard Annex #15²⁶.

For example:

Listing 329: show_identifiers.adb

```
1 procedure Show_Identifiers is
2   --      ^^^^^^^^^^^^^^^
3   --      identifier
4
5   type New_Integer is new
6   --      ^^^^^^^^^^^^^
```

(continues on next page)

²³ <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>

²⁴ <https://en.wikipedia.org/wiki/ASCII>

²⁵ https://en.wikipedia.org/wiki/Universal_Coded_Character_Set

²⁶ <https://unicode.org/reports/tr15/>

(continued from previous page)

```

7      -- identifier
8      Integer;
9      ^^^^^
10     identifier
11
12     Something_Important : New_Integer;
13     ^^^^^^^^^^^^^^^^^
14     identifier
15     ^^^^^^^^^^^^^
16     identifier
17 begin
18     null;
19 end Show_Identifiers;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Names.Identifiers
MD5: e427d3e5fe5f549df593b5e5941cf2ba

In this example, we see the following identifiers: `Show_Identifiers` (subprogram), `New_Integer` (type), **`Integer`** (type), and `Something_Important` (object).

Operator symbols

The set of operator symbols that we can use is restricted to the following symbols or reserved words specified in the Ada language:

Operator kind	Operators
Logical operators	and, or, xor
Relational operators	<code>=</code> , <code>/=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code>
Binary adding operators	<code>+</code> , <code>-</code> , <code>&</code>
Unary adding operators	<code>+</code> , <code>-</code>
multiplying opertors	<code>*</code> , <code>/</code> , mod, rem
Highest precedence operators	<code>**</code> , abs, not

In the Ada Reference Manual

- [4.5 Operators and Expression Evaluation²⁷](#)

Other kinds of names

In addition to direct names, we have the following kinds of names: *explicit dereferences* (page 911), indexed components, slices, selected components, attribute references, *type conversions* (page 319), function calls, character literals, *qualified expressions* (page 338), *generalized references* (page 1030), and target name.

Let's see an example of some of them:

Listing 330: `show_other_names.adb`

```

1  pragma Ada_2022;
2
3  procedure Show_Other_Names is

```

(continues on next page)

²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-5.html>

(continued from previous page)

```

4
5  type Integer_Access is
6     access Integer;
7
8  type Integer_Array is array
9     (Positive range <>) of Integer;
10
11 type New_Integer is new
12     Integer;
13
14 function Zero
15     return New_Integer is
16     (0);
17
18 subtype Sub_Integer is
19     Integer;
20
21 type Rec is record
22     Val : Integer := 0;
23 end record;
24
25 type ABC_Enum is
26     ('A', 'B', 'C');
27
28 IA : Integer_Access := new Integer;
29 Arr : Integer_Array (1 .. 5) :=
30     (others => 0);
31 R : Rec;
32 NI : New_Integer;
33 SI : Sub_Integer;
34 E : ABC_Enum := 'A';
35 begin
36     R.Val := IA.all;
37     --      ^^^^^^
38     --  explicit dereference
39
40     R.Val := Arr (1);
41     --      ^^^^^^
42     --  indexed component
43
44     Arr (1 .. 2) := Arr (3 .. 4);
45     --      ^^^^^^^^^^^^^
46     --                  slice
47
48     Arr (1 .. 2) := (others => R.Val);
49     --      ^^^^^
50     --  selected component
51
52     R.Val := Arr'Size;
53     --      ^^^^^^^
54     --  attribute reference
55
56     NI := New_Integer (IA.all);
57     --      ^^^^^^^^^^^^^^^^^
58     --  type conversion
59
60     NI := Zero;
61     --      ^^^
62     --  function call
63
64     E := 'A';

```

(continues on next page)

(continued from previous page)

```

65  --      ^^
66  --  character literal
67
68  IA.all := Sub_Integer (R.Val);
69  --      ~~~~~
70  --      qualified expression
71
72  R.Val := @ + 1;
73  --      ^
74  --  target name
75  --
76  --  equivalent to:
77  --      R.Val := R.Val + 1;
78
79  end Show_Other_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Names.Other_Names
MD5: 8063a4c9ff7a01ff7a69454fae096089

In this example, we see instances of the following kinds of names:

- explicit dereference: IA.all;
- indexed components: Arr (1);
- slices: Arr (1 .. 2), Arr (3 .. 4);
- selected components: R.Val;
- attribute references: Arr'Size;
- type conversions: New_Integer (IA.all);
- function calls: Zero;
- character literals: 'A';
- qualified expressions: Sub_Integer (R.Val);
- target name: @.

In the Ada Reference Manual

- 4.1 Name²⁸
- 4.1.1 Indexed Components²⁹
- 4.1.2 Slices³⁰
- 4.1.3 Selected Components³¹
- 4.1.4 Attributes³²
- 4.1.5 User-Defined References³³
- 4.6 Type Conversions³⁴
- 4.7 Qualified Expressions³⁵
- 5.2.1 Target Name Symbols³⁶

25.1.2 Objects

The term *object* may be misleading for readers that have a strong background in object-oriented programming. Moreover, its meaning can vary depending on the context. Therefore, it's important to define what we mean by *objects* when focusing on Ada programming.

In computer science, the term *object*³⁷ can refer to a piece of data stored in memory — but it can also refer to a table or a form in a database. Also, even when we define the term *object* as data in memory, we can still classify programming languages as *object-based*³⁸ or *object-oriented*³⁹ languages.

Important

In object-oriented programming, an object belongs to a *class* of objects. In Ada, objects of this kind are called *tagged* objects. Note, however, that we can have objects that don't belong to a class of objects: those are called *untagged* objects.

In the context of Ada programming, an object is an "entity that contains a value, and is either a constant or a variable" — according to the Ada Reference Manual. In other words, any constants or variables that we declare in Ada source code are objects. In addition, there are other examples of objects that don't originate from object declarations:

Listing 331: show_objects.adb

```

1  procedure Show_Objects is
2
3      type New_Integer is new
4          Integer;
5
6      type Integer_Array is
7          array (Positive range <>) of
8              Integer;
9
10     procedure Dummy (Obj : Integer)
11         is null;
12     --          ^^
13     --          object
14
15     task type TT is
16         entry Start (Id : Integer);
17         --          ^^
18         --          object
19     end TT;
20
21     task body TT is
22     begin
23         accept Start (Id : Integer) do
24             null;
25         end Start;

```

(continues on next page)

²⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>
²⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html>
³⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html>
³¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-3.html>
³² <http://www.ada-auth.org/standards/22rm/html/RM-4-1-4.html>
³³ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html>
³⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-6.html>
³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-7.html>
³⁶ <http://www.ada-auth.org/standards/22rm/html/RM-5-2-1.html>
³⁷ [https://en.wikipedia.org/wiki/Object_\(computer_science\)](https://en.wikipedia.org/wiki/Object_(computer_science))
³⁸ https://en.wikipedia.org/wiki/Object-based_language
³⁹ https://en.wikipedia.org/wiki/Object-oriented_programming

(continued from previous page)

```

26  end TT;
27
28  function Add_One (V : Integer)
29      ^
30      -- view of an object
31      return Integer is
32  begin
33      return V + 1;
34      ^^^^^
35      -- object
36  end Add_One;
37
38  Arr : Integer_Array (1 .. 10);
39      ^^^^^^^^^^^^^^^^^
40      -- object
41
42  NI : New_Integer;
43  begin
44      Arr (1 .. 3) := (others => 1);
45      ^^^^^
46      -- object
47      ^^^^^^^^^^^^^
48      -- object
49
50      NI := New_Integer (Arr (1));
51      ^^^^^^^^^^^^^^^^^
52      -- object
53
54      for I in Arr'Range loop
55          ^
56          -- object
57
58          Arr (I) := Add_One (Arr (I));
59          ^^^^^
60          -- object
61      end loop;
62  end Show_Objects;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Examples
MD5: edf9eab70ec0ecce90ef71591324ac94

As we can see in this code example a formal parameter of a subprogram or an entry is also an object — in addition, so are *value conversions* (page 319), the result returned by a function, the result of evaluating an *aggregate* (page 516), loop parameters, *arrays* (page 564), or the slices of arrays objects, or the components of composite objects.

Other examples of objects include:

- the object created via a *view conversion* (page 328);
- a *dereference* (page 911) of an *access-to-variable* (page 925) value;
- the return object of a function;
- a choice parameter of an *exception handler* (page 146).

In the Ada Reference Manual

- 3.3 Objects and Named Numbers⁴⁰

Constant and variable objects

Objects can be classified as constant and variable objects. When declaring objects, the distinction is clear:

Listing 332: show_objects.adb

```

1 procedure Show_Objects is
2   Const : constant Integer := 42;
3   Var   : Integer := 0;
4 begin
5   null;
6 end Show_Objects;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Declaration_Examples
 MD5: 16b4d9546e9c05443ced05c7f6608cc9

In this example, Const is a constant object, while Var is a variable object.

In addition to this, constant objects include:

- the *discriminant component* (page 458) of a variable discriminant;
- a formal parameter or generic formal object of mode **in**.

On the other hand, variable objects include:

- the object created via a *view conversion* (page 328) of a variable;
- a *dereference* (page 911) of an *access-to-variable* (page 925) value.

For example:

Listing 333: show_objects.adb

```

1 procedure Show_Objects is
2
3   type Device (Id : Positive) is
4   record
5     Value : Integer;
6   end record;
7
8   type Device_Access is
9     access all Device;
10
11   Dev : aliased Device (99);
12   --      ^^
13   -- Discriminant `Id` is a
14   -- constant object.
15   --
16   -- `Dev` is a variable object,
17   -- though.
18
19   Dev_Acc : Device_Access := Dev'Access;
20
21   procedure Process (D : Device) is
22     null;
23     --      ^
24     --      constant object
25 begin
26   Dev.Value := 0;
```

(continues on next page)

⁴⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-3.html>

(continued from previous page)

```

27  --  ^^^^^
28  --  variable object
29
30  Dev_Acc.all.Value := 1;
31  --  ^^^^^^^
32  --  variable object
33  end Show_Objects;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_Examples
 MD5: c0d1386a37e5ed31f0d3163fadbb1b30

In this example, we see that `Dev` is a variable object, while its `Id` discriminant is a constant object. In addition, the `Dev_Acc.all` dereference is a variable object. Finally, the `in` parameter of procedure `Process` is a constant object.

In the Ada Reference Manual

- 3.3 Objects and Named Numbers⁴¹
- 3.3.1 Object Declarations⁴²

View of an object

As we've just seen, an object can be either constant or variable. In addition, the *view* of an object is classified as constant or variable as well.

Before we start, note that the classification of an object as constant or variable doesn't directly imply how its view is classified. You may, for example, expect that a constant object has a constant view, but this is not necessarily the case, as we discuss in this section. (In fact, a constant object only has a constant view if it doesn't have a part that has a variable view.)

A part of an object has a variable view if it is of *immutably limited type* (page 1089), *controlled type* (page 1121), *private type* (page 312), or private extension. In that sense, if any of those parts with variable view exist in a constant object, then we say that the *whole object* has a variable view. Only if a constant object doesn't have *any* parts with variable view, then this object has a constant view.

In contrast, variable objects always have a variable view.

Let's see an example:

Listing 334: devices.ads

```

1  package Devices is
2
3      type Device_Settings is
4      record
5          Started : Boolean;
6      end record;
7
8      type Device (Id : Positive) is
9      private;
10
11     function Init (Id : Positive)

```

(continues on next page)

⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-3.html>

⁴² <http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html>

(continued from previous page)

```

12         return Device;
13
14 private
15
16     type Device (Id : Positive) is
17         null record;
18
19     function Init (Id : Positive)
20         return Device is
21         (Device'(Id => Id));
22
23 end Devices;

```

Listing 335: show_object_view.adb

```

1 with Devices; use Devices;
2
3 procedure Show_Object_View is
4     Dev      : constant Device := Init (5);
5     -- Constant object with
6     -- variable view.
7
8     Default  : constant Device_Settings
9         := (Started => False);
10    -- Constant object with
11    -- constant view.
12
13    Settings : Device_Settings;
14
15 begin
16     Settings := (Started => True);
17 end Show_Object_View;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Object_View
MD5: b9a56ee937e71c728bac116f21d98742

In this example, both `Default_S` and `Dev` are constant objects. However, they have different views: while `Default_S` has a constant view because it doesn't have any parts with variable view, `Dev` has a variable view because it's a private type. Finally, as expected, `Settings` has a variable view because it's a variable object.

Named numbers

In addition to objects, we can have named numbers. Those aren't objects, but rather *names* (page 279) that we assign to numeric values. For example:

Listing 336: show_named_number.adb

```

1 procedure Show_Named_Number is
2
3     Pi : constant := 3.1415926535;
4
5 begin
6     null;
7 end Show_Named_Number;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Objects.Named_Number
 MD5: ee6808bb7ecb7fef687831f53a8b6668

In this example, `Pi` is a named number.

A named number is always known at compilation time. Also, it doesn't have a type associated with it. In fact, its type is called universal real or universal integer — depending on the number being a real or integer number. (In this specific case, `Pi` is a universal real number.) We talk about *universal types* (page 302) later on in this chapter and about *universal real and integer types* (page 629) in another chapter.

i In the Ada Reference Manual

- 3.3.2 Number Declarations⁴³

25.1.3 Scalar Types

In general terms, scalar types are the most basic types that we can get. As we know, we can classify them as follows:

Category	Discrete	Numeric
Enumeration	Yes	No
Integer	Yes	Yes
Real	No	Yes

Many attributes exist for scalar types. For example, we can use the `Image` and `Value` attributes to convert between a given type and a string type. The following table presents the main attributes for scalar types:

Category	Attribute	Returned value
Ranges	<code>First</code>	First value of the discrete subtype's range.
	<code>Last</code>	Last value of the discrete subtype's range.
	<code>Range</code>	Range of the discrete subtype (corresponds to <code>Subtype'First .. Subtype'Last</code>).
Iterators	<code>Pred</code>	Predecessor of the input value.
	<code>Succ</code>	Successor of the input value.
Comparison	<code>Min</code>	Minimum of two values.
	<code>Max</code>	Maximum of two values.
String conversion	<code>Image</code>	String representation of the input value.
	<code>Value</code>	Value of a subtype based on input string.

We already discussed some of these attributes in the Introduction to Ada course (in the sections about *range and related attributes* (page 76) and *image attribute* (page 13)). In this section, we'll discuss some aspects that have been left out of the previous course.

i In the Ada Reference Manual

- 3.5 Scalar types⁴⁴

⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-3-3-2.html>

⁴⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

Ranges

We've seen that the `First` and `Last` attributes can be used with discrete types. Those attributes are also available for real types. Here's an example using the **Float** type and a subtype of it:

Listing 337: show_first_last_real.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_First_Last_Real is
4   subtype Norm is Float range 0.0 .. 1.0;
5 begin
6   Put_Line ("Float'First: " & Float'First'Image);
7   Put_Line ("Float'Last:  " & Float'Last'Image);
8   Put_Line ("Norm'First:  " & Norm'First'Image);
9   Put_Line ("Norm'Last:   " & Norm'Last'Image);
10 end Show_First_Last_Real;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Ranges_Real_Types
MD5: 89745a94fbdc41a2880ba14e50401acb

Runtime output

```

Float'First: -3.40282E+38
Float'Last:  3.40282E+38
Norm'First:  0.00000E+00
Norm'Last:   1.00000E+00
```

This program displays the first and last values of both the **Float** type and the `Norm` subtype. In the case of the **Float** type, we see the full range, while for the `Norm` subtype, we get the values we used in the declaration of the subtype (i.e. 0.0 and 1.0).

Predecessor and Successor

We can use the `Pred` and `Succ` attributes to get the predecessor and successor of a specific value. For discrete types, this is simply the next discrete value. For example, `Pred (2)` is 1 and `Succ (2)` is 3. Let's look at a complete source-code example:

Listing 338: show_succ_pred_discrete.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Succ_Pred_Discrete is
4   type State is (Idle, Started,
5                 Processing, Stopped);
6
7   Machine_State : constant State := Started;
8
9   I : constant Integer := 2;
10 begin
11   Put_Line ("State           : "
12           & Machine_State'Image);
13   Put_Line ("State'Pred (Machine_State): "
14           & State'Pred (Machine_State)'Image);
15   Put_Line ("State'Succ (Machine_State): "
16           & State'Succ (Machine_State)'Image);
17   Put_Line ("-----");
18
19   Put_Line ("I           : "
```

(continues on next page)

(continued from previous page)

```

20         & I'Image);
21     Put_Line ("Integer'Pred (I): "
22             & Integer'Pred (I)'Image);
23     Put_Line ("Integer'Succ (I): "
24             & Integer'Succ (I)'Image);
25 end Show_Succ_Pred_Discrete;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Show_Succ_Pred_Discrete
MD5: e11d0f50105864fdc1594b3bb72d927e

Runtime output

```

State                : STARTED
State'Pred (Machine_State): IDLE
State'Succ (Machine_State): PROCESSING
-----
I                    : 2
Integer'Pred (I): 1
Integer'Succ (I): 3

```

In this example, we use the Pred and Succ attributes for a variable of enumeration type (State) and a variable of **Integer** type.

We can also use the Pred and Succ attributes with real types. In this case, however, the value we get depends on the actual type we're using:

- for fixed-point types, the value is calculated using the smallest value (Small), which is derived from the declaration of the fixed-point type;
- for floating-point types, the value used in the calculation depends on representation constraints of the actual target machine.

Let's look at this example with a decimal type (Decimal) and a floating-point type (My_Float):

Listing 339: show_succ_pred_real.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Succ_Pred_Real is
4      subtype My_Float is
5          Float range 0.0 .. 0.5;
6
7      type Decimal is
8          delta 0.1 digits 2
9          range 0.0 .. 0.5;
10
11     D : Decimal;
12     N : My_Float;
13 begin
14     Put_Line ("---- DECIMAL ----");
15     Put_Line ("Small: " & Decimal'Small'Image);
16     Put_Line ("----- Succ -----");
17     D := Decimal'First;
18     loop
19         Put_Line (D'Image);
20         D := Decimal'Succ (D);
21
22         exit when D = Decimal'Last;
23     end loop;

```

(continues on next page)

(continued from previous page)

```

24   Put_Line ("----- Pred -----");
25
26   D := Decimal'Last;
27   loop
28       Put_Line (D'Image);
29       D := Decimal'Pred (D);
30
31       exit when D = Decimal'First;
32   end loop;
33   Put_Line ("=====");
34
35   Put_Line ("---- MY_FLOAT ----");
36   Put_Line ("----- Succ -----");
37   N := My_Float'First;
38   for I in 1 .. 5 loop
39       Put_Line (N'Image);
40       N := My_Float'Succ (N);
41   end loop;
42   Put_Line ("----- Pred -----");
43
44   for I in 1 .. 5 loop
45       Put_Line (N'Image);
46       N := My_Float'Pred (N);
47   end loop;
48 end Show_Succ_Pred_Real;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Show_Succ_Pred_Real
MD5: f426d6539c3ce863101f1e6afb21c08f

Runtime output

```

---- DECIMAL ----
Small: 1.0000000000000000E-01
----- Succ -----
0.0
0.1
0.2
0.3
0.4
----- Pred -----
0.5
0.4
0.3
0.2
0.1
=====
---- MY_FLOAT ----
----- Succ -----
0.00000E+00
1.40130E-45
2.80260E-45
4.20390E-45
5.60519E-45
----- Pred -----
7.00649E-45
5.60519E-45
4.20390E-45
2.80260E-45
1.40130E-45

```

As the output of the program indicates, the smallest value (see `Decimal'Small` in the example) is used to calculate the previous and next values of `Decimal` type.

In the case of the `My_Float` type, the difference between the current and the previous or next values is `1.40130E-45` (or 2^{-149}) on a standard PC.

Scalar To String Conversion

We've seen that we can use the `Image` and `Value` attributes to perform conversions between values of a given subtype and a string:

Listing 340: `show_image_value_attr.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Image_Value_Attr is
4   I : constant Integer := Integer'Value ("42");
5 begin
6   Put_Line (I'Image);
7 end Show_Image_Value_Attr;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Image_Value_Attr`
MD5: `9daa13b1f05511fac7e108eb9b8eefa7`

Runtime output

42

The `Image` and `Value` attributes are used for the `String` type specifically. In addition to them, there are also attributes for different string types — namely `Wide_String` and `Wide_Wide_String`. This is the complete list of available attributes:

Conversion type	Attribute	String type
Conversion to string	<code>Image</code>	<code>String</code>
	<code>Wide_Image</code>	<code>Wide_String</code>
	<code>Wide_Wide_Image</code>	<code>Wide_Wide_String</code>
Conversion to subtype	<code>Value</code>	<code>String</code>
	<code>Wide_Value</code>	<code>Wide_String</code>
	<code>Wide_Wide_Value</code>	<code>Wide_Wide_String</code>

We discuss more about `Wide_String` and `Wide_Wide_String` in [another section](#) (page 581).

Width attribute

When converting a value to a string by using the `Image` attribute, we get a string with variable width. We can assess the maximum width of that string for a specific subtype by using the `Width` attribute. For example, `Integer'Width` gives us the maximum width returned by the `Image` attribute when converting a value of `Integer` type to a string of `String` type.

This attribute is useful when we're using bounded strings in our code to store the string returned by the `Image` attribute. For example:

Listing 341: show_width_attr.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Strings;          use Ada.Strings;
3  with Ada.Strings.Bounded;
4
5  procedure Show_Width_Attr is
6      package B_Str is new
7          Ada.Strings.Bounded.Generic_Bounded_Length
8              (Max => Integer'Width);
9      use B_Str;
10
11     Str_I : Bounded_String;
12
13     I : constant Integer := 42;
14     J : constant Integer := 103;
15 begin
16     Str_I := To_Bounded_String (I'Image);
17     Put_Line ("Value: "
18             & To_String (Str_I));
19     Put_Line ("String Length: "
20             & Length (Str_I)'Image);
21     Put_Line ("----");
22
23     Str_I := To_Bounded_String (J'Image);
24     Put_Line ("Value: "
25             & To_String (Str_I));
26     Put_Line ("String Length: "
27             & Length (Str_I)'Image);
28 end Show_Width_Attr;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Scalar_Types.Width_Attr
MD5: 82cff0cf4fecfdecce3020135cf98fd2

Runtime output

```

Value:      42
String Length: 3
----
Value:      103
String Length: 4

```

In this example, we're storing the string returned by `Image` in the `Str_I` variable of `Bounded_String` type.

Similar to the `Image` and `Value` attributes, the `Width` attribute is also available for string types other than **String**. In fact, we can use:

- the `Wide_Width` attribute for strings returned by `Wide_Image`; and
- the `Wide_Wide_Width` attribute for strings returned by `Wide_Wide_Image`.

Other attributes

The `Base` attribute is specific for numeric types. We discuss this topic *in another chapter* (page 644).

25.1.4 Enumerations

We've introduced enumerations back in the *Introduction to Ada course* (page 51). In this section, we'll discuss a few useful features of enumerations, such as enumeration renaming, enumeration overloading and representation clauses.

In the Ada Reference Manual

- [3.5.1 Enumeration Types](#)⁴⁵

Enumerations as functions

If you have used programming language such as C in the past, you're familiar with the concept of enumerations being constants with integer values. In Ada, however, enumerations are not integers. In fact, they're actually parameterless functions! Let's consider this example:

Listing 342: days.ads

```

1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     -- Essentially, we're declaring
8     -- these functions:
9     --
10    -- function Mon return Day;
11    -- function Tue return Day;
12    -- function Wed return Day;
13    -- function Thu return Day;
14    -- function Fri return Day;
15    -- function Sat return Day;
16    -- function Sun return Day;
17
18 end Days;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_As_Function
MD5: fa3e58b58edffa5a3e04b060a7f8cb8b

In the package Days, we're declaring the enumeration type Day. When we do this, we're essentially declaring seven parameterless functions, one for each enumeration. For example, the Mon enumeration corresponds to **function Mon return Day**. You can see all seven function declarations in the comments of the example above.

Note that this has no direct relation to how an Ada compiler generates machine code for enumeration. Even though enumerations are parameterless functions, a typical Ada compiler doesn't generate function calls for code that deals with enumerations.

Enumeration renaming

The idea that enumerations are parameterless functions can be used when we want to rename enumerations. For example, we could rename the enumerations of the Day type like this:

⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-1.html>

Listing 343: enumeration_example.ads

```

1 package Enumeration_Example is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     function Monday    return Day renames Mon;
8     function Tuesday   return Day renames Tue;
9     function Wednesday return Day renames Wed;
10    function Thursday  return Day renames Thu;
11    function Friday    return Day renames Fri;
12    function Saturday  return Day renames Sat;
13    function Sunday    return Day renames Sun;
14
15 end Enumeration_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: e2e12bb3bfc0b6e94769ced9a4b80f9

Now, we can use both Monday or Mon to refer to Monday of the Day type:

Listing 344: show_renaming.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Enumeration_Example; use Enumeration_Example;
3
4 procedure Show_Renaming is
5     D1 : constant Day := Mon;
6     D2 : constant Day := Monday;
7 begin
8     if D1 = D2 then
9         Put_Line ("D1 = D2");
10        Put_Line (Day'Image (D1)
11                & " = "
12                & Day'Image (D2));
13    end if;
14 end Show_Renaming;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: 2d7177def2c9e9fb11c7dc5e036c3be3

Runtime output

```

D1 = D2
MON = MON
```

When running this application, we can confirm that D1 is equal to D2. Also, even though we've assigned Monday to D2 (instead of Mon), the application displays Mon = Mon, since Monday is just another name to refer to the actual enumeration (Mon).

** Hint**

If you just want to have a single (renamed) enumeration visible in your application — and make the original enumeration invisible —, you can use a separate package. For

example:

Listing 345: enumeration_example.ads

```

1 package Enumeration_Example is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7 end Enumeration_Example;
```

Listing 346: enumeration_renaming.ads

```

1 with Enumeration_Example;
2
3 package Enumeration_Renaming is
4
5     subtype Day is Enumeration_Example.Day;
6
7     function Monday    return Day renames
8         Enumeration_Example.Mon;
9     function Tuesday   return Day renames
10        Enumeration_Example.Tue;
11    function Wednesday return Day renames
12        Enumeration_Example.Wed;
13    function Thursday   return Day renames
14        Enumeration_Example.Thu;
15    function Friday     return Day renames
16        Enumeration_Example.Fri;
17    function Saturday   return Day renames
18        Enumeration_Example.Sat;
19    function Sunday     return Day renames
20        Enumeration_Example.Sun;
21
22 end Enumeration_Renaming;
```

Listing 347: show_renaming.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Enumeration_Renaming;
4 use Enumeration_Renaming;
5
6 procedure Show_Renaming is
7     D1 : constant Day := Monday;
8 begin
9     Put_Line (Day'Image (D1));
10 end Show_Renaming;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Renaming
MD5: 87fe75026f0fc118921eaae45fe55a8a

Runtime output

MON

Note that the call to Put_Line still display Mon instead of Monday.

Enumeration overloading

Enumerations can be overloaded. In simple terms, this means that the same name can be used to declare an enumeration of different types. A typical example is the declaration of colors:

Listing 348: colors.ads

```

1 package Colors is
2
3   type Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   type Primary_Color is
17     (Red,
18      Green,
19      Blue);
20
21 end Colors;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
MD5: b808f90d9164f044b6b7a8931863726f

Note that we have Red as an enumeration of type Color and of type Primary_Color. The same applies to Green and Blue. Because Ada is a strongly-typed language, in most cases, the enumeration that we're referring to is clear from the context. For example:

Listing 349: red_colors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Red_Colors is
5   C1 : constant Color      := Red;
6   -- Using Red from Color
7
8   C2 : constant Primary_Color := Red;
9   -- Using Red from Primary_Color
10 begin
11   if C1 = Red then
12     Put_Line ("C1 = Red");
13   end if;
14   if C2 = Red then
15     Put_Line ("C2 = Red");
16   end if;
17 end Red_Colors;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
MD5: dd590eab88164773e974e748d77a51af

Runtime output

```
C1 = Red
C2 = Red
```

When assigning Red to C1 and C2, it is clear that, in the first case, we're referring to Red of Color type, while in the second case, we're referring to Red of the Primary_Color type. The same logic applies to comparisons such as the one in `if C1 = Red`: because the type of C1 is defined (Color), it's clear that the Red enumeration is the one of Color type.

Enumeration subtypes

Note that enumeration overloading is not the same as enumeration subtypes. For example, we could define the following subtype:

Listing 350: colors-shades.ads

```
1 package Colors.Shades is
2
3     subtype Blue_Shades is
4         Colors range Blue .. Darkblue;
5
6 end Colors.Shades;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Overloading
MD5: 9c13508bda487cae02dbf8b403271540
```

In this case, Blue of Blue_Shades and Blue of Colors are the same enumeration.

Enumeration ambiguities

A situation where enumeration overloading might lead to ambiguities is when we use them in ranges. For example:

Listing 351: colors.ads

```
1 package Colors is
2
3     type Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     type Primary_Color is
17         (Red,
18          Green,
19          Blue);
20
21 end Colors;
```

Listing 352: color_loop.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5 begin
6     for C in Red .. Blue loop
7         -- ~~~~~
8         -- ERROR: range is ambiguous!
9         Put_Line (Color'Image (C));
10    end loop;
11 end Color_Loop;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities
MD5: 82d0d3f28f1faf6b296a4f44db71f41b

Build output

```

color_loop.adb:6:17: error: ambiguous bounds in range of iteration
color_loop.adb:6:17: error: possible interpretations:
color_loop.adb:6:17: error: type "Primary_Color" defined at colors.ads:16
color_loop.adb:6:17: error: type "Color" defined at colors.ads:3
color_loop.adb:6:17: error: ambiguous bounds in discrete range
color_loop.adb:9:30: error: expected type "Color" defined at colors.ads:3
color_loop.adb:9:30: error: found type "Primary_Color" defined at colors.ads:16
gprbuild: *** compilation phase failed

```

Here, it's not clear whether the range in the loop is of Color type or of Primary_Color type. Therefore, we get a compilation error for this code example. The next line in the code example — the one with the call to Put_Line — gives us a hint about the developer's intention to refer to the Color type. In this case, we can use qualification — for example, Color' (Red) — to resolve the ambiguity:

Listing 353: color_loop.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Color_Loop is
5 begin
6     for C in Color'(Red) .. Color'(Blue) loop
7         Put_Line (Color'Image (C));
8     end loop;
9 end Color_Loop;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities
MD5: c3e946d330bb6aed258bcd005a540794

Runtime output

```

RED
DARKRED
LIME
FORESTGREEN
GREEN

```

(continues on next page)

(continued from previous page)

```
DARKGREEN  
BLUE
```

Note that, in the case of ranges, we can also rewrite the loop by using a range declaration:

Listing 354: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2 with Colors;      use Colors;  
3  
4 procedure Color_Loop is  
5 begin  
6   for C in Color range Red .. Blue loop  
7     Put_Line (Color'Image (C));  
8   end loop;  
9 end Color_Loop;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Ambiguities  
MD5: 23f8db4fcb5710f7bda6b511234e0448
```

Runtime output

```
RED  
DARKRED  
LIME  
FORESTGREEN  
GREEN  
DARKGREEN  
BLUE
```

Alternatively, Color **range** Red .. Blue could be used in a subtype declaration, so we could rewrite the example above using a subtype (such as Red_To_Blue) in the loop:

Listing 355: color_loop.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2 with Colors;      use Colors;  
3  
4 procedure Color_Loop is  
5   subtype Red_To_Blue is Color range Red .. Blue;  
6 begin  
7   for C in Red_To_Blue loop  
8     Put_Line (Color'Image (C));  
9   end loop;  
10 end Color_Loop;
```

Position and Internal Code

As we've said above, a typical Ada compiler doesn't generate function calls for code that deals with enumerations. On the contrary, each enumeration has values associated with it, and the compiler uses those values instead.

Each enumeration has:

- a position value, which is a natural value indicating the position of the enumeration in the enumeration type; and
- an internal code, which, by default, in most cases, is the same as the position value.

Also, by default, the value of the first position is zero, the value of the second position is one, and so on. We can see this by listing each enumeration of the Day type and displaying the value of the corresponding position:

Listing 356: days.ads

```

1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7 end Days;
```

Listing 357: show_days.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Days;        use Days;
3
4 procedure Show_Days is
5 begin
6     for D in Day loop
7         Put_Line (Day'Image (D)
8                 & " position      = "
9                 & Integer'Image (Day'Pos (D)));
10        Put_Line (Day'Image (D)
11                & " internal code = "
12                & Integer'Image
13                (Day'Enum_Rep (D)));
14    end loop;
15 end Show_Days;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Enumerations.Enumeration_Values
MD5: d6c5cb99b9770893b7277c470f40e805

Runtime output

```

MON position      = 0
MON internal code = 0
TUE position      = 1
TUE internal code = 1
WED position      = 2
WED internal code = 2
THU position      = 3
THU internal code = 3
FRI position      = 4
FRI internal code = 4
SAT position      = 5
SAT internal code = 5
SUN position      = 6
SUN internal code = 6
```

Note that this application also displays the internal code, which, in this case, is equivalent to the position value for all enumerations.

We may, however, change the internal code of an enumeration using a representation clause. We discuss this topic [in another section](#) (page 352).

25.1.5 Universal and Root Types

Previously, in the section about *scalar types* (page 288), we said that scalar types are the most basic types that we can get. However, Ada has the concept of universal and root types, which could be considered *more basic* than scalar types. In fact, universal and root types are underlying scalar types used by the language designers to define the language semantics. In this section, we briefly introduce this topic.

Universal Types

The Ada standard defines four universal types:

1. universal integer types
2. universal real types
3. universal fixed types
4. universal access types

The first three are numeric types, and we discuss them in detail later on *in another chapter* (page 629). The last one is used for *anonymous access types* (page 997).

Universal types aren't types we can use directly, but rather via specific languages constructs. In this sense, we cannot derive from universal types, but only make use of them indirectly.

For instance, if we declare *named numbers* (page 287) using a real value, we're indirectly using a universal real type. If we declare another named number using an expression, the computation is performed based on the universal types of the elements of that expression:

Listing 358: show_universal_real_integer.ads

```

1 package Show_Universal_Real_Integer is
2
3   Pi      : constant := 3.1415926535;
4           --
5           --          universal real type
6
7   Two_Pi  : constant := Pi * 2.0;
8           --
9           --          operation on
10          --          universal real type
11
12   N       : constant := 10;
13           --
14           --          universal integer type
15
16   N_10    : constant := N * 10;
17           --
18           --          operation on
19           --          universal integer type
20
21 end Show_Universal_Real_Integer;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Universal_Real_Integer
 MD5: c9f002461d8ee7f11f2c42a33691f30d

In this example, the expression `Pi * 2.0` is computed using universal real types, while the expression `N * 10` is computed using universal integer types.

Similarly, for anonymous access types, the equality operator uses universal access types for the comparison:

Listing 359: show_universal_access.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Universal_Access is
4   I : aliased Integer;
5   A : access Integer := I'Access;
6   B : access Integer := I'Access;
7 begin
8   if A = B then
9     Put_Line ("A = B");
10  else
11    Put_Line ("A /= B");
12  end if;
13 end Show_Universal_Access;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Universal_Access

MD5: e6a37de980cc3b2c19e36baa3a51c329

Runtime output

A = B

In this example, both A and B are variables of anonymous access types. Because the type isn't a known named type, the equality operation = uses the universal access type for the comparison.

In the Ada Reference Manual

- 3.3.2 Number Declarations⁴⁶
- 4.5.2 Relational Operators and Membership Tests⁴⁷

Root Types

The root types can be found on a level above the universal types. In this category, we can find the same numeric types that we have for universal types, namely the root real, root integer and root fixed types.

The term *root* is used in the context of type derivation. In fact, the root type is the first type that we derive all other types from. In other words, if we declare an integer range as a new type, that type is derived from the root integer type. Similarly, if we declare a new floating-point type, that type is derived from the root real type. For example:

Listing 360: show_root_integer_real.ads

```

1 package Show_Root_Integer_Real is
2
3   type Score is range 0 .. 10;
4   -- Type Score is derived from
5   -- the root integer type.
6

```

(continues on next page)

⁴⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-3-2.html>

⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html>

(continued from previous page)

```

7  type Real_Score is
8      digits 10 range 0.0 .. 10.0;
9      -- Type Real_Score is derived from
10     -- the root real type.
11
12 end Show_Root_Integer_Real;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Root_Integer_Real
 MD5: 619ecddeab6fd751cc1af9daa8794a25

Here, `Score` and `Real_Score` are derived from the root integer and real types, respectively. Note that the derivation is always implicit, as we cannot write something like `type Score is new Root_Integer range 0 .. 10` or `type Real_Score is new Root_Real digits 10 range 0.0 .. 10.0`.

In contrast, if we derive from an existing floating-point or integer type defined by the Ada standard, we're not deriving directly from the root types:

Listing 361: show_standard_derivation.ads

```

1  package Show_Standard_Derivation is
2
3      type Score is new Integer
4          range 0 .. 10;
5      -- Type Score is derived from
6      -- the Integer type.
7
8      type Real_Score is new Float
9          range 0.0 .. 10.0;
10     -- Type Real_Score is derived from
11     -- the Float type.
12
13 end Show_Standard_Derivation;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Universal_And_Root_Types.Standard_Integer_Float_Derivation
 MD5: d32261966e1f1ae9626336f57ab16d89

In this case, we're explicitly deriving from the standard Ada types `Integer` and `Float`, which, on their turn, are derived from the root integer and root real types, respectively.

** For further reading...**

Ada also has the concept of *base types* (page 643), which *sounds* similar to the concept of the root type. However, the focus of each one is different: while the root type refers to the derivation tree of a type, the base type refers to the constraints of a type.

We discuss base types and the *Base attribute* (page 644) in another chapter.

25.1.6 Definite and Indefinite Subtypes

Indefinite types were mentioned back in the *Introduction to Ada course* (page 81). In this section, we'll recapitulate and extend on both definite and indefinite types.

Definite types are the basic kind of types we commonly use when programming applications. For example, we can only declare variables of definite types; otherwise, we get a compilation error. Interestingly, however, to be able to explain what definite types are, we need to first discuss indefinite types.

Indefinite types include:

- unconstrained arrays;
- record types with unconstrained discriminants without defaults.

Let's see some examples of indefinite types:

Listing 362: unconstrained_types.ads

```

1 package Unconstrained_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Simple_Record (Extended : Boolean) is
7     record
8         V : Integer;
9         case Extended is
10             when False =>
11                 null;
12             when True =>
13                 V_Float : Float;
14         end case;
15     end record;
16
17 end Unconstrained_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↳ Indefinite_Types
MD5: e569dc73150b834c9315b14d46c0ac79

In this example, both `Integer_Array` and `Simple_Record` are indefinite types.

As we've just mentioned, we cannot declare variable of indefinite types:

Listing 363: using_unconstrained_type.adb

```

1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Using_Unconstrained_Type is
4
5     A : Integer_Array;
6
7     R : Simple_Record;
8
9 begin
10     null;
11 end Using_Unconstrained_Type;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↳ Indefinite_Types
MD5: 806d4ec64b911a9978ad30fa45a6df10

Build output

```
using_unconstrained_type.adb:5:08: error: unconstrained subtype not allowed (need_
↳ initialization)
using_unconstrained_type.adb:5:08: error: provide initial value or explicit array_
↳ bounds
using_unconstrained_type.adb:7:08: error: unconstrained subtype not allowed (need_
↳ initialization)
using_unconstrained_type.adb:7:08: error: provide initial value or explicit_
↳ discriminant values
using_unconstrained_type.adb:7:08: error: or give default discriminant values for_
↳ type "Simple_Record"
gprbuild: *** compilation phase failed
```

As we can see when we try to build this example, the compiler complains about the declaration of A and R because we're trying to use indefinite types to declare variables. The main reason we cannot use indefinite types here is that the compiler needs to know at this point how much memory it should allocate. Therefore, we need to provide the information that is missing. In other words, we need to change the declaration so the type becomes definite. We can do this by either declaring a definite type or providing constraints in the variable declaration. For example:

Listing 364: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Using_Unconstrained_Type is
4
5     subtype Integer_Array_5 is
6         Integer_Array (1 .. 5);
7
8     A1 : Integer_Array_5;
9     A2 : Integer_Array (1 .. 5);
10
11     subtype Simple_Record_Ext is
12         Simple_Record (Extended => True);
13
14     R1 : Simple_Record_Ext;
15     R2 : Simple_Record (Extended => True);
16
17 begin
18     null;
19 end Using_Unconstrained_Type;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↳ Indefinite_Types
MD5: f8e192537f42eea0ebc7873bdad898f1
```

In this example, we declare the `Integer_Array_5` subtype, which is definite because we're constraining it to a range from 1 to 5, thereby defining the information that was missing in the indefinite type `Integer_Array`. Because we now have a definite type, we can use it to declare the `A1` variable. Similarly, we can use the indefinite type `Integer_Array` directly in the declaration of `A2` by specifying the previously unknown range.

Similarly, in this example, we declare the `Simple_Record_Ext` subtype, which is definite because we're initializing the record discriminant `Extended`. We can therefore use it in the declaration of the `R1` variable. Alternatively, we can simply use the indefinite type `Simple_Record` and specify the information required for the discriminants. This is what we do in the declaration of the `R2` variable.

Although we cannot use indefinite types directly in variable declarations, they're very useful to generalize algorithms. For example, we can use them as parameters of a subprogram:

Listing 365: show_integer_array.ads

```

1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array (A : Integer_Array);

```

Listing 366: show_integer_array.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Integer_Array (A : Integer_Array)
4 is
5 begin
6     for I in A'Range loop
7         Put_Line (Positive'Image (I)
8                 & ": "
9                 & Integer'Image (A (I)));
10    end loop;
11    Put_Line ("-----");
12 end Show_Integer_Array;

```

Listing 367: using_unconstrained_type.adb

```

1 with Unconstrained_Types; use Unconstrained_Types;
2 with Show_Integer_Array;
3
4 procedure Using_Unconstrained_Type is
5     A_5 : constant Integer_Array (1 .. 5) :=
6         (1, 2, 3, 4, 5);
7     A_10 : constant Integer_Array (1 .. 10) :=
8         (1, 2, 3, 4, 5, others => 99);
9 begin
10     Show_Integer_Array (A_5);
11     Show_Integer_Array (A_10);
12 end Using_Unconstrained_Type;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↳ Indefinite_Types
 MD5: 3f744fa5921a55865bc5361ec4c6eb88

Runtime output

```

1: 1
2: 2
3: 3
4: 4
5: 5
-----
1: 1
2: 2
3: 3
4: 4
5: 5
6: 99
7: 99
8: 99
9: 99
10: 99
-----

```

In this particular example, the compiler doesn't know a priori which range is used for the A parameter of Show_Integer_Array. It could be a range from 1 to 5 as used for variable A_5 of the Using_Unconstrained_Type procedure, or it could be a range from 1 to 10 as used for variable A_10, or it could be anything else. Although the parameter A of Show_Integer_Array is unconstrained, both calls to Show_Integer_Array — in Using_Unconstrained_Type procedure — use constrained objects.

Note that we could call the Show_Integer_Array procedure above with another unconstrained parameter. For example:

Listing 368: show_integer_array_header.ads

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array_Header
4   (AA : Integer_Array;
5    HH : String);
```

Listing 369: show_integer_array_header.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Show_Integer_Array;
3
4 procedure Show_Integer_Array_Header
5   (AA : Integer_Array;
6    HH : String)
7 is
8 begin
9   Put_Line (HH);
10  Show_Integer_Array (AA);
11 end Show_Integer_Array_Header;
```

Listing 370: using_unconstrained_type.adb

```
1 with Unconstrained_Types; use Unconstrained_Types;
2
3 with Show_Integer_Array_Header;
4
5 procedure Using_Unconstrained_Type is
6   A_5 : constant Integer_Array (1 .. 5) :=
7     (1, 2, 3, 4, 5);
8   A_10 : constant Integer_Array (1 .. 10) :=
9     (1, 2, 3, 4, 5, others => 99);
10 begin
11   Show_Integer_Array_Header (A_5,
12                               "First example");
13   Show_Integer_Array_Header (A_10,
14                               "Second example");
15 end Using_Unconstrained_Type;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
↪ Indefinite_Types
MD5: dd09f8c4089c6ad4c18410879f80f731

Runtime output

```
First example
1: 1
2: 2
3: 3
```

(continues on next page)

(continued from previous page)

```

4:  4
5:  5
-----
Second example
1:  1
2:  2
3:  3
4:  4
5:  5
6:  99
7:  99
8:  99
9:  99
10: 99
-----

```

In this case, we're calling the `Show_Integer_Array` procedure with another unconstrained parameter (the `AA` parameter). However, although we could have a long *chain* of procedure calls using indefinite types in their parameters, we still use a (definite) object at the beginning of this chain. For example, for the `A_5` object, we have this chain:

```

A_5

==> Show_Integer_Array_Header (AA => A_5,
                               ...);

==> Show_Integer_Array (A => AA);

```

Therefore, at this specific call to `Show_Integer_Array`, even though `A` is declared as a parameter of indefinite type, the actual argument is of definite type because `A_5` is constrained — and, thus, of definite type.

Note that we can declare variables based on parameters of indefinite type. For example:

Listing 371: `show_integer_array_plus.ads`

```

1 with Unconstrained_Types; use Unconstrained_Types;
2
3 procedure Show_Integer_Array_Plus
4   (A : Integer_Array;
5    V : Integer);

```

Listing 372: `show_integer_array_plus.adb`

```

1 with Show_Integer_Array;
2
3 procedure Show_Integer_Array_Plus
4   (A : Integer_Array;
5    V : Integer)
6 is
7   A_Plus : Integer_Array (A'Range);
8 begin
9   for I in A_Plus'Range loop
10    A_Plus (I) := A (I) + V;
11  end loop;
12  Show_Integer_Array (A_Plus);
13 end Show_Integer_Array_Plus;

```


Listing 373: using_unconstrained_type.adb

```

1 with Unconstrained_Types; use Unconstrained_Types;
2
3 with Show_Integer_Array_Plus;
4
5 procedure Using_Unconstrained_Type is
6   A_5 : constant Integer_Array (1 .. 5) :=
7     (1, 2, 3, 4, 5);
8 begin
9   Show_Integer_Array_Plus (A_5, 5);
10 end Using_Unconstrained_Type;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↳ Indefinite_Types
 MD5: e58ae62272ff0b27c5f6e171c88a6880

Runtime output

```

1: 6
2: 7
3: 8
4: 9
5: 10
-----

```

In the `Show_Integer_Array_Plus` procedure, we're declaring `A_Plus` based on the range of `A`, which is itself of indefinite type. However, since the object passed as an argument to `Show_Integer_Array_Plus` must have a constraint, `A_Plus` will also be constrained. For example, in the call to `Show_Integer_Array_Plus` using `A_5` as an argument, the declaration of `A_Plus` becomes `A_Plus : Integer_Array (1 .. 5);`. Therefore, it becomes clear that the compiler needs to allocate five elements for `A_Plus`.

We'll see later how definite and indefinite types apply to formal parameters.

** In the Ada Reference Manual**

- 3.3 Objects and Named Numbers⁴⁸

25.1.7 Incomplete types

Incomplete types — as the name suggests — are types that have missing information in their declaration. This is a simple example:

```
type Incomplete;
```

Because this type declaration is incomplete, we need to provide the missing information at some later point. Consider the incomplete type `R` in the following example:

Listing 374: incomplete_type_example.ads

```

1 package Incomplete_Type_Example is
2
3   type R;
4   -- Incomplete type declaration!

```

(continues on next page)

⁴⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-3.html>

(continued from previous page)

```

5
6  type R is record
7      I : Integer;
8  end record;
9      -- type R is now complete!
10
11 end Incomplete_Type_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Incomplete_Types
MD5: 5ca250595f2b0cc101df286ab319982f

The first declaration of type R is incomplete. However, in the second declaration of R, we specify that R is a record. By providing this missing information, we're completing the type declaration of R.

It's also possible to declare an incomplete type in the private part of a package specification and its complete form in the package body. Let's rewrite the example above accordingly:

Listing 375: incomplete_type_example.ads

```

1 package Incomplete_Type_Example is
2
3 private
4
5     type R;
6     -- Incomplete type declaration!
7
8 end Incomplete_Type_Example;
```

Listing 376: incomplete_type_example.adb

```

1 package body Incomplete_Type_Example is
2
3     type R is record
4         I : Integer;
5     end record;
6     -- type R is now complete!
7
8 end Incomplete_Type_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Incomplete_Types_2
MD5: fd2f0301b4a63887add1cb2093692ddb

A typical application of incomplete types is to create linked lists using [access types](#) (page 881) based on those incomplete types. This kind of type is called a recursive type. For example:

Listing 377: linked_list_example.ads

```

1 package Linked_List_Example is
2
3     type Integer_List;
4
5     type Next is access Integer_List;
6
7     type Integer_List is record
8         I : Integer;
```

(continues on next page)

(continued from previous page)

```

9      N : Next;
10     end record;
11
12 end Linked_List_Example;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Incomplete_Types.Linked_List_Example
MD5: b2d3a048473d498bbe691bc6e38ca1e9

Here, the `N` component of `Integer_List` is essentially giving us access to the next element of `Integer_List` type. Because the `Next` type is both referring to the `Integer_List` type and being used in the declaration of the `Integer_List` type, we need to start with an incomplete declaration of the `Integer_List` type and then complete it after the declaration of `Next`.

Incomplete types are useful to declare *mutually dependent types* (page 448), as we'll see later on. Also, we can also have formal incomplete types, as we'll discuss later.

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations⁴⁹

25.1.8 Type view

Ada distinguishes between the partial and the full view of a type. The full view is a type declaration that contains all the information needed by the compiler. For example, the following declaration of type `R` represents the full view of this type:

Listing 378: full_view.ads

```

1 package Full_View is
2
3     -- Full view of the R type:
4     type R is record
5         I : Integer;
6     end record;
7
8 end Full_View;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Full_View
MD5: d37792287d08f9aa3d32499e233516df

As soon as we start applying encapsulation and information hiding — via the **private** key-word — to a specific type, we are introducing a partial view and making only that view compile-time visible to clients. Doing so requires us to introduce the private part of the package (unless already present). For example:

Listing 379: partial_full_views.ads

```

1 package Partial_Full_Views is
2
3     -- Partial view of the R type:
4     type R is private;

```

(continues on next page)

⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html>

(continued from previous page)

```

5
6 private
7
8   -- Full view of the R type:
9   type R is record
10     I : Integer;
11   end record;
12
13 end Partial_Full_Views;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Partial_Full_View
MD5: b0cf748e43b23ea6c845e283c4266ff3

As indicated in the example, the **type R is private** declaration is the partial view of the R type, while the **type R is record** [...] declaration in the private part of the package is the full view.

Although the partial view doesn't contain the full type declaration, it contains very important information for the users of the package where it's declared. In fact, the partial view of a private type is all that users actually need to know to effectively use this type, while the full view is only needed by the compiler.

In the previous example, the partial view indicates that R is a private type, which means that, even though users cannot directly access any information stored in this type — for example, read the value of the I component of R —, they can use the R type to declare objects. For example:

Listing 380: main.adb

```

1 with Partial_Full_Views; use Partial_Full_Views;
2
3 procedure Main is
4   -- Partial view of R indicates that
5   -- R exists as a private type, so we
6   -- can declare objects of this type:
7   C : R;
8 begin
9   -- But we cannot directly access any
10  -- information declared in the full
11  -- view of R:
12  --
13  -- C.I := 42;
14  --
15  null;
16 end Main;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Partial_Full_View
MD5: 05bc9a75406d0a46f6d009d97885d010

In many cases, the restrictions applied to the partial and full views must match. For example, if we declare a limited type in the full view of a private type, its partial view must also be limited:

Listing 381: limited_private_example.ads

```

1 package Limited_Private_Example is
2

```

(continues on next page)

(continued from previous page)

```

3  -- Partial view must be limited,
4  -- since the full view is limited.
5  type R is limited private;
6
7 private
8
9  type R is limited record
10     I : Integer;
11 end record;
12
13 end Limited_Private_Example;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Limited_Private
MD5: 23d01b93fe052a500c8ca6ff76a2fd51

There are, however, situations where the full view may contain additional requirements that aren't mentioned in the partial view. For example, a type may be declared as non-tagged in the partial view, but, at the same time, be tagged in the full view:

Listing 382: tagged_full_view_example.ads

```

1 package Tagged_Full_View_Example is
2
3  -- Partial view using non-tagged type:
4  type R is private;
5
6 private
7
8  -- Full view using tagged type:
9  type R is tagged record
10     I : Integer;
11 end record;
12
13 end Tagged_Full_View_Example;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Tagged_Full_View
MD5: 0ff9142b1ee086695b98b72a9d0f50ac

In this case, from a user's perspective, the R type is non-tagged, so that users cannot use any object-oriented programming features for this type. In the package body of Tagged_Full_View_Example, however, this type is tagged, so that all object-oriented programming features are available for subprograms of the package body that make use of this type. Again, the partial view of the private type contains the most important information for users that want to declare objects of this type.

** In the Ada Reference Manual**

- 7.3 Private Types and Private Extensions⁵⁰

⁵⁰ <http://www.ada-auth.org/standards/22rm/html/RM-7-3.html>

Non-Record Private Types

Although it's very common to declare private types as record types, this is not the only option. In fact, we could declare any type in the full view — scalars, for example —, so we could declare a "private integer" type:

Listing 383: private_integers.ads

```

1 package Private_Integers is
2
3   -- Partial view of private Integer type:
4   type Private_Integer is private;
5
6 private
7
8   -- Full view of private Integer type:
9   type Private_Integer is new Integer;
10
11 end Private_Integers;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: f1fcbed95e0f66a6f67d1bfd9ba9df1c

This code compiles as expected, but isn't very useful. We can improve it by adding operators to it, for example:

Listing 384: private_integers.ads

```

1 package Private_Integers is
2
3   -- Partial view of private Integer type:
4   type Private_Integer is private;
5
6   function "+" (Left, Right : Private_Integer)
7     return Private_Integer;
8
9 private
10
11   -- Full view of private Integer type:
12   type Private_Integer is new Integer;
13
14 end Private_Integers;
```

Listing 385: private_integers.adb

```

1 package body Private_Integers is
2
3   function "+" (Left, Right : Private_Integer)
4     return Private_Integer
5   is
6     Res : constant Integer :=
7       Integer (Left) + Integer (Right);
8     -- Note that we're converting Left
9     -- and Right to Integer, which calls
10    -- the "+" operator of the Integer
11    -- type. Writing "Left + Right" would
12    -- have called the "+" operator of
13    -- Private_Integer, which leads to
14    -- recursive calls, as this is the
15    -- operator we're currently in.
```

(continues on next page)

(continued from previous page)

```
16   begin
17       return Private_Integer (Res);
18   end "+";
19
20 end Private_Integers;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: ac161cb5debfde16465c45949cf682d7

Now, let's use the new operator in a test application:

Listing 386: show_private_integers.adb

```
1 with Private_Integers; use Private_Integers;
2
3 procedure Show_Private_Integers is
4     A, B : Private_Integer;
5 begin
6     A := A + B;
7 end Show_Private_Integers;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Integer
MD5: 5933779ce5f0802b448df96c42e65a8d

Build output

```
show_private_integers.adb:4:07: warning: variable "B" is read but never assigned [-
↳gnatwv]
show_private_integers.adb:6:09: warning: "A" may be referenced before it has a
↳value [enabled by default]
```

In this example, we use the + operator as if we were adding two common integer variables of **Integer** type.

Unconstrained Types

There are, however, some limitations: we cannot use unconstrained types such as arrays or even discriminants for arrays in the same way as we did for scalars. For example, the following declarations won't work:

Listing 387: private_arrays.ads

```
1 package Private_Arrays is
2
3     type Private_Unconstrained_Array is private;
4
5     type Private_Constrained_Array
6         (L : Positive) is private;
7
8 private
9
10    type Integer_Array is
11        array (Positive range <>) of Integer;
12
13    type Private_Unconstrained_Array is
14        array (Positive range <>) of Integer;
```

(continues on next page)

(continued from previous page)

```

15
16 type Private_Constrained_Array
17   (L : Positive) is
18   array (1 .. 2) of Integer;
19
20   -- NOTE: using an array type fails as well:
21   --
22   -- type Private_Constrained_Array
23   --   (L : Positive) is
24   --     Integer_Array (1 .. L);
25
26 end Private_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: b873c2d381c159532b429101e4533c05

Build output

```

private_arrays.ads:13:09: error: full view of "Private_Unconstrained_Array" not
↳ compatible with declaration at line 3
private_arrays.ads:13:09: error: one is constrained, the other unconstrained
private_arrays.ads:17:07: error: elementary or array type cannot have discriminants
gprbuild: *** compilation phase failed

```

Completing the private type with an unconstrained array type in the full view is not allowed because clients could expect, according to their view, to declare objects of the type. But doing so would not be allowed according to the full view. So this is another case of the partial view having to present clients with a sufficiently *true* view of the type's capabilities.

One solution is to rewrite the declaration of **Private_Constrained_Array** using a record type:

Listing 388: private_arrays.ads

```

1 package Private_Arrays is
2
3   type Private_Constrained_Array
4     (L : Positive) is private;
5
6 private
7
8   type Integer_Array is
9     array (Positive range <>) of Integer;
10
11   type Private_Constrained_Array
12     (L : Positive) is
13   record
14     Arr : Integer_Array (1 .. 2);
15   end record;
16
17 end Private_Arrays;

```

Listing 389: declare_private_array.adb

```

1 with Private_Arrays; use Private_Arrays;
2
3 procedure Declare_Private_Array is
4   Arr : Private_Constrained_Array (5);
5 begin

```

(continues on next page)

(continued from previous page)

```
6   null;  
7 end Declare_Private_Array;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: 3830721499a59d85efddd4989aa7c288

Now, the code compiles fine — but we had to use a record type in the full view to make it work.

Another solution is to make the private type indefinite. In this case, the client's partial view would be consistent with a completion as an indefinite type in the private part:

Listing 390: private_arrays.ads

```
1 package Private_Arrays is  
2  
3   type Private_Constrained_Array (<>) is  
4     private;  
5  
6   function Init  
7     (L : Positive)  
8     return Private_Constrained_Array;  
9  
10 private  
11  
12   type Private_Constrained_Array is  
13     array (Positive range <>) of Integer;  
14  
15 end Private_Arrays;
```

Listing 391: private_arrays.adb

```
1 package body Private_Arrays is  
2  
3   function Init  
4     (L : Positive)  
5     return Private_Constrained_Array  
6   is  
7     PCA : Private_Constrained_Array (1 .. L);  
8   begin  
9     return PCA;  
10  end Init;  
11  
12 end Private_Arrays;
```

Listing 392: declare_private_array.adb

```
1 with Private_Arrays; use Private_Arrays;  
2  
3 procedure Declare_Private_Array is  
4   Arr : Private_Constrained_Array := Init (5);  
5 begin  
6   null;  
7 end Declare_Private_Array;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_View.Private_Array
MD5: cd170a1e44fffb93314776a68f1cb413

Build output

```
private_arrays.adb:7:07: warning: variable "PCA" is read but never assigned [-gnatwv]
```

The bounds for the object's declaration come from the required initial value when an object is declared. In this case, we initialize the object with a call to the `Init` function.

25.1.9 Type conversion

An important operation when dealing with objects of different types is type conversion, which we already discussed in the *Introduction to Ada course* (page 56). In fact, we can convert an object `Obj_X` of an *operand* type `X` to a similar, closely related *target* type `Y` by simply indicating the target type: `Y (Obj_X)`. In this section, we discuss type conversions for different kinds of types.

Ada distinguishes between two kinds of conversion: value conversion and view conversion. The main difference is the way how the operand (argument) of the conversion is evaluated:

- in a value conversion, the operand is evaluated as an *expression* (page 725);
- in a view conversion, the operand is evaluated as a name.

In other words, we cannot use expressions such as `2 * A` in a view conversion, but only `A`. In a value conversion, we could use both forms.

In the Ada Reference Manual

- [4.6 Type Conversions](#)⁵¹

Value conversion

Value conversions are possible for various types. In this section, we see some examples, starting with types derived from scalar types up to array conversions.

Root and derived types

Let's start with the conversion between a scalar type and its derived types. For example, we can convert back-and-forth between the **Integer** type and the derived `Int` type:

Listing 393: custom_integers.ads

```
1 package Custom_Integers is
2
3     type Int is new Integer
4       with Dynamic_Predicate => Int /= 0;
5
6     function Double (I : Integer)
7       return Integer is
8       (I * 2);
9
10 end Custom_Integers;
```

Listing 394: show_conversion.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
```

(continues on next page)

⁵¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-6.html>

(continued from previous page)

```

3
4 procedure Show_Conversion is
5   Int_Var      : Int      := 1;
6   Integer_Var  : Integer := 2;
7 begin
8   -- Int to Integer conversion
9   Integer_Var := Integer (Int_Var);
10
11   Put_Line ("Integer_Var : "
12            & Integer_Var'Image);
13
14   -- Int to Integer conversion
15   -- as an actual parameter
16   Integer_Var := Double (Integer (Int_Var));
17
18   Put_Line ("Integer_Var : "
19            & Integer_Var'Image);
20
21   -- Integer to Int conversion
22   -- using an expression
23   Int_Var      := Int (Integer_Var * 2);
24
25   Put_Line ("Int_Var : "
26            & Int_Var'Image);
27 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_Conversion
 MD5: 7cd324f308edc34de3bc4bccce63f1ee

Runtime output

```

Integer_Var : 1
Integer_Var : 2
Int_Var :    4

```

In the `Show_Conversion` procedure from this example, we first convert from `Int` to **Integer**. Then, we do the same conversion while providing the resulting value as an actual parameter for the `Double` function. Finally, we convert the `Integer_Var * 2` expression from **Integer** to `Int`.

Note that the converted value must conform to any constraints that the target type might have. In the example above, `Int` has a predicate that dictates that its value cannot be zero. This (dynamic) predicate is checked at runtime, so an exception is raised if it fails:

Listing 395: show_conversion.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
3
4 procedure Show_Conversion is
5   Int_Var      : Int;
6   Integer_Var  : Integer;
7 begin
8   Integer_Var := 0;
9   Int_Var     := Int (Integer_Var);
10
11   Put_Line ("Int_Var : "
12            & Int_Var'Image);
13

```

(continues on next page)

(continued from previous page)

```
13 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Root_Derived_Type_Conversion
 MD5: 4150cdfd4c1fed39fa1728a77fa599f

Runtime output

```
raised ADA.Assertions.Assertion_Error : Dynamic_Predicate failed at show_
↳ conversion.adb:9
```

In this case, the conversion from **Integer** to `Int` fails because, while zero is a valid integer value, it doesn't obey `Int`'s predicate.

Numeric type conversion

A typical conversion is the one between integer and floating-point values. For example:

Listing 396: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Conversion is
4   F : Float := 1.0;
5   I : Integer := 2;
6 begin
7   I := Integer (F);
8
9   Put_Line ("I : "
10             & I'Image);
11
12   I := 4;
13   F := Float (I);
14
15   Put_Line ("F : "
16             & F'Image);
17 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_Conversion
 MD5: f64649c786377617b0bc9ff49475ba55

Runtime output

```
I : 1
F : 4.00000E+00
```

Also, we can convert between fixed-point types and floating-point or integer types:

Listing 397: fixed_point_defs.ads

```
1 package Fixed_Point_Defs is
2   S : constant := 32;
3   Exp : constant := 15;
4   D : constant := 2.0 ** (-S + Exp + 1);
```

(continues on next page)

(continued from previous page)

```

5
6   type TQ15_31 is delta D
7     range -1.0 * 2.0 ** Exp ..
8           1.0 * 2.0 ** Exp - D;
9
10  pragma Assert (TQ15_31'Size = S);
11 end Fixed_Point_Defs;

```

Listing 398: show_conversion.adb

```

1  with Fixed_Point_Defs; use Fixed_Point_Defs;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  procedure Show_Conversion is
5    F : Float;
6    FP : TQ15_31;
7    I : Integer;
8  begin
9    FP := TQ15_31 (10.25);
10   I := Integer (FP);
11
12   Put_Line ("FP : "
13             & FP'Image);
14   Put_Line ("I : "
15             & I'Image);
16
17   I := 128;
18   FP := TQ15_31 (I);
19   F := Float (FP);
20
21   Put_Line ("FP : "
22             & FP'Image);
23   Put_Line ("F : "
24             & F'Image);
25 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Numeric_Type_Conversion
 MD5: 70714ba396b03469397b982e00299561

Runtime output

```

FP : 10.25000
I : 10
FP : 128.00000
F : 1.28000E+02

```

As we can see in the examples above, converting between different numeric types works in all directions. (Of course, rounding is applied when converting from floating-point to integer types, but this is expected.)

Enumeration conversion

We can also convert between an enumeration type and a type derived from it:

Listing 399: custom_enumerations.ads

```

1 package Custom_Enumerations is
2
3     type Priority is (Low, Mid, High);
4
5     type Important_Priority is new
6         Priority range Mid .. High;
7
8 end Custom_Enumerations;
```

Listing 400: show_conversion.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Custom_Enumerations; use Custom_Enumerations;
3
4 procedure Show_Conversion is
5     P : Priority := Low;
6     IP : Important_Priority := High;
7 begin
8     P := Priority (IP);
9
10    Put_Line ("P: "
11              & P'Image);
12
13    P := Mid;
14    IP := Important_Priority (P);
15
16    Put_Line ("IP: "
17              & IP'Image);
18 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Enumeration_Type_↵
 Conversion
 MD5: b1e42cbd8b57291d3b3a9968c41efdd7

Runtime output

```
P: HIGH
IP: MID
```

In this example, we have the `Priority` type and the derived type `Important_Priority`. As expected, the conversion works fine when the converted value is in the range of the target type. If not, an exception is raised:

Listing 401: show_conversion.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Custom_Enumerations; use Custom_Enumerations;
3
4 procedure Show_Conversion is
5     P : Priority;
6     IP : Important_Priority;
7 begin
8     P := Low;
9     IP := Important_Priority (P);
10
11    Put_Line ("IP: "
12              & IP'Image);
```

(continues on next page)

(continued from previous page)

```
13 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Enumeration_Type_Conversion
MD5: 6bbc777d4b44023bf572ca5dc6c2b4f8

Build output

```
show_conversion.adb:9:10: warning: value not in range of type "Important_Priority"
↳ defined at custom_enumerations.ads:5 [enabled by default]
show_conversion.adb:9:10: warning: Constraint_Error will be raised at run time
↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_conversion.adb:9 range check failed
```

In this example, an exception is raised because Low is not in the Important_Priority type's range.

Array conversion

Similarly, we can convert between array types. For example, if we have the array type Integer_Array and its derived type Derived_Integer_Array, we can convert between those array types:

Listing 402: custom_arrays.ads

```
1 package Custom_Arrays is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Derived_Integer_Array is new
7         Integer_Array;
8
9 end Custom_Arrays;
```

Listing 403: show_conversion.adb

```
1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Custom_Arrays; use Custom_Arrays;
3
4 procedure Show_Conversion is
5     subtype Common_Range is Positive range 1 .. 3;
6
7     AI : Integer_Array (Common_Range);
8     AI_D : Derived_Integer_Array (Common_Range);
9 begin
10     AI_D := [1, 2, 3];
11     AI := Integer_Array (AI_D);
12
13     Put_Line ("AI: "
14               & AI'Image);
15
16     AI := [4, 5, 6];
17     AI_D := Derived_Integer_Array (AI);
```

(continues on next page)

(continued from previous page)

```

18
19     Put_Line ("AI_D: "
20               & AI_D'Image);
21 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_Conversion
 MD5: 72cdf4850bec78893b6985b0c7ef02b9

Runtime output

```

AI:
[ 1,  2,  3]
AI_D:
[ 4,  5,  6]

```

Note that both arrays must have the same number of components in order for the conversion to be successful. (Sliding is fine, though.) In this example, both arrays have the same range: `Common_Range`.

We can also convert between array types that aren't derived one from the other. As long as the components and the index subtypes are of the same type, the conversion between those types is possible. To be more precise, these are the requirements for the array conversion to be accepted:

- The component types must be the same type.
- The index types (or subtypes) must be the same or, at least, convertible.
- The dimensionality of the arrays must be the same.
- The bounds must be compatible (but not necessarily equal).

Converting between different array types can be very handy, especially when we're dealing with array types that were not declared in the same package. For example:

Listing 404: `custom_arrays_1.ads`

```

1 package Custom_Arrays_1 is
2
3     type Integer_Array_1 is
4       array (Positive range <>) of Integer;
5
6     type Float_Array_1 is
7       array (Positive range <>) of Float;
8
9 end Custom_Arrays_1;

```

Listing 405: `custom_arrays_2.ads`

```

1 package Custom_Arrays_2 is
2
3     type Integer_Array_2 is
4       array (Positive range <>) of Integer;
5
6     type Float_Array_2 is
7       array (Positive range <>) of Float;
8
9 end Custom_Arrays_2;

```


Listing 406: show_conversion.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Custom_Arrays_1; use Custom_Arrays_1;
3  with Custom_Arrays_2; use Custom_Arrays_2;
4
5  procedure Show_Conversion is
6      subtype Common_Range is Positive range 1 .. 3;
7
8      AI_1 : Integer_Array_1 (Common_Range);
9      AI_2 : Integer_Array_2 (Common_Range);
10     AF_1 : Float_Array_1 (Common_Range);
11     AF_2 : Float_Array_2 (Common_Range);
12 begin
13     AI_2 := [1, 2, 3];
14     AI_1 := Integer_Array_1 (AI_2);
15
16     Put_Line ("AI_1: "
17              & AI_1'Image);
18
19     AI_1 := [4, 5, 6];
20     AI_2 := Integer_Array_2 (AI_1);
21
22     Put_Line ("AI_2: "
23              & AI_2'Image);
24
25     -- ERROR: Cannot convert arrays whose
26     --         components have different types:
27     --
28     -- AF_1 := Float_Array_1 (AI_1);
29     --
30     -- Instead, use array aggregate where each
31     -- component is converted from integer to
32     -- float:
33     --
34     AF_1 := [for I in AF_1'Range =>
35              Float (AI_1 (I))];
36
37     Put_Line ("AF_1: "
38              & AF_1'Image);
39
40     AF_2 := Float_Array_2 (AF_1);
41
42     Put_Line ("AF_2: "
43              & AF_2'Image);
44 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Array_Type_

 Conversion
 MD5: 42b89fa5fe1f20af26b5da4586cea8e8

Runtime output

```

AI_1:
[ 1,  2,  3]
AI_2:
[ 4,  5,  6]
AF_1:
[ 4.00000E+00,  5.00000E+00,  6.00000E+00]

```

(continues on next page)

(continued from previous page)

```
AF_2:
[ 4.00000E+00,  5.00000E+00,  6.00000E+00]
```

As we can see in this example, the fact that `Integer_Array_1` and `Integer_Array_2` have the same component type (**Integer**) allows us to convert between them. The same applies to the `Float_Array_1` and `Float_Array_2` types.

A conversion is not possible when the component types don't match. Even though we can convert between integer and floating-point types, we cannot convert an array of integers to an array of floating-point directly. Therefore, we cannot write a statement such as `AF_1 := Float_Array_1 (AI_1);`.

However, when the components don't match, we can of course implement the array conversion by converting the individual components. For the example above, we used an iterated component association in an array aggregate: `[for I in AF_1'Range => Float (AI_1 (I))];`. (We discuss this topic later *in another chapter* (page 531).)

We may also encounter array types originating from the instantiation of generic packages. In this case as well, we can use array conversions. Consider the following generic package:

Listing 407: custom_arrays.ads

```
1 generic
2   type T is private;
3 package Custom_Arrays is
4   type T_Array is
5     array (Positive range <>) of T;
6 end Custom_Arrays;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Generic_Array_Type_Conversion
 MD5: 8b3a963a1292a90d99d83c6d81ce3995

We could instantiate this generic package and reuse parts of the previous code example:

Listing 408: show_conversion.adb

```
1 with Ada.Text_IO;   use Ada.Text_IO;
2 with Custom_Arrays;
3
4 procedure Show_Conversion is
5   package CA_Int_1 is
6     new Custom_Arrays (T => Integer);
7   package CA_Int_2 is
8     new Custom_Arrays (T => Integer);
9
10    subtype Common_Range is Positive range 1 .. 3;
11
12    AI_1 : CA_Int_1.T_Array (Common_Range);
13    AI_2 : CA_Int_2.T_Array (Common_Range);
14 begin
15   AI_2 := [1, 2, 3];
16   AI_1 := CA_Int_1.T_Array (AI_2);
17
18   Put_Line ("AI_1: "
19             & AI_1'Image);
20
21   AI_1 := [4, 5, 6];
22   AI_2 := CA_Int_2.T_Array (AI_1);
```

(continues on next page)

(continued from previous page)

```
23
24     Put_Line ("AI_2: "
25               & AI_2'Image);
26 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Generic_Array_Type_Conversion
MD5: f5348b3bed5cbd93dab44394358e1ce6

Runtime output

```
AI_1:
[ 1,  2,  3]
AI_2:
[ 4,  5,  6]
```

As we can see in this example, each of the instantiated `CA_Int_1` and `CA_Int_2` packages has a `T_Array` type. Even though these `T_Array` types have the same name, they're actually completely unrelated types. However, we can still convert between them in the same way as we did in the previous code examples.

View conversion

As mentioned before, view conversions just allow names to be converted. Thus, we cannot use expressions in this case.

Note that a view conversion never changes the value during the conversion. We could say that a view conversion is simply making us *view* an object from a different angle. The object itself is still the same for both the original and the target types.

For example, consider this package:

Listing 409: some_tagged_types.ads

```
1 package Some_Tagged_Types is
2
3     type T is tagged record
4         A : Integer;
5     end record;
6
7     type T_Derived is new T with record
8         B : Float;
9     end record;
10
11     Obj : T_Derived;
12
13 end Some_Tagged_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Types_View
MD5: 2e18ba972682f1ae1d38e38842fde48e

Here, `Obj` is an object of type `T_Derived`. When we *view* this object, we notice that it has two components: `A` and `B`. However, we could *view* this object as being of type `T`. From that perspective, this object only has one component: `A`. (Note that changing the perspective doesn't change the object itself.) Therefore, a view conversion from `T_Derived` to `T` just makes us *view* the object `Obj` from a different angle.

In this sense, a view conversion changes the view of a given object to the target type's view, both in terms of components that exist and operations that are available. It doesn't really change anything at all in the value itself.

There are basically two kinds of view conversions: the ones using tagged types and the ones using untagged types. We discuss these kinds of conversion in this section.

View conversion of tagged types

A conversion between tagged types is a view conversion. Let's consider a typical code example that declares one, two and three-dimensional points:

Listing 410: points.ads

```

1 package Points is
2
3   type Point_1D is tagged record
4     X : Float;
5   end record;
6
7   procedure Display (P : Point_1D);
8
9   type Point_2D is new Point_1D with record
10     Y : Float;
11   end record;
12
13   procedure Display (P : Point_2D);
14
15   type Point_3D is new Point_2D with record
16     Z : Float;
17   end record;
18
19   procedure Display (P : Point_3D);
20
21 end Points;
```

Listing 411: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_1D) is
6   begin
7     Put_Line ("(X => " & P.X'Image & ")");
8   end Display;
9
10  procedure Display (P : Point_2D) is
11  begin
12    Put_Line ("(X => " & P.X'Image
13              & ", Y => " & P.Y'Image & ")");
14  end Display;
15
16  procedure Display (P : Point_3D) is
17  begin
18    Put_Line ("(X => " & P.X'Image
19              & ", Y => " & P.Y'Image
20              & ", Z => " & P.Z'Image & ")");
21  end Display;
22
23 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_Conversion
MD5: 0acc05ae2310ab4ba038dfdb6bae0495

We can use the types from the Points package and convert between each other:

Listing 412: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Points;      use Points;
3
4 procedure Show_Conversion is
5     P_1D : Point_1D;
6     P_3D : Point_3D;
7 begin
8     P_3D := (X => 0.1, Y => 0.5, Z => 0.3);
9     P_1D := Point_1D (P_3D);
10
11     Put ("P_3D : ");
12     Display (P_3D);
13
14     Put ("P_1D : ");
15     Display (P_1D);
16 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Tagged_Type_Conversion
MD5: fb8e07c8f2399cfae935179d8f413150

Runtime output

```
P_3D : (X => 1.00000E-01, Y => 5.00000E-01, Z => 3.00000E-01)
P_1D : (X => 1.00000E-01)
```

In this example, as expected, we're able to convert from the Point_3D type (which has three components) to the Point_1D type, which has only one component.

View conversion of untagged types

For untagged types, a view conversion is the one that happens when we have an object of an untagged type as an actual parameter for a formal **in out** or **out** parameter.

Let's see a code example. Consider the following simple procedure:

Listing 413: double.ads

```
1 procedure Double (X : in out Float);
```

Listing 414: double.adb

```
1 procedure Double (X : in out Float) is
2 begin
3     X := X * 2.0;
4 end Double;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
 MD5: 31f4409d9faeaf213c5940de65eeb014

The Double procedure has an **in out** parameter of **Float** type. We can call this procedure using an integer variable I as the actual parameter. For example:

Listing 415: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Double;
3
4 procedure Show_Conversion is
5   I : Integer;
6 begin
7   I := 2;
8   Put_Line ("I : "
9             & I'Image);
10
11   -- Calling Double with
12   -- Integer parameter:
13   Double (Float (I));
14   Put_Line ("I : "
15             & I'Image);
16 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
 MD5: 2256d3c120d569789dcd4c9959ed9d0f

Runtime output

```

I : 2
I : 4
```

In this case, the **Float** (I) conversion in the call to Double creates a temporary floating-point variable. This is the same as if we had written the following code:

Listing 416: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Double;
3
4 procedure Show_Conversion is
5   I : Integer;
6 begin
7   I := 2;
8   Put_Line ("I : "
9             & I'Image);
10
11   declare
12     F : Float := Float (I);
13   begin
14     Double (F);
15     I := Integer (F);
16   end;
17   Put_Line ("I : "
18             & I'Image);
19 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Untagged_Type_View_Conversion
MD5: 3b90caf789952710ece42141a7b60968

Runtime output

```
I : 2
I : 4
```

In this sense, the view conversion that happens in Double (**Float** (I)) can be considered syntactic sugar, as it allows us to elegantly write two conversions in a single statement.

Implicit conversions

Implicit conversions are only possible when we have a type T and a subtype S related to the T type. For example:

Listing 417: custom_integers.ads

```
1 package Custom_Integers is
2
3     type Int is new Integer
4       with Dynamic_Predicate => Int /= 0;
5
6     subtype Sub_Int_1 is Integer
7       with Dynamic_Predicate => Sub_Int_1 /= 0;
8
9     subtype Sub_Int_2 is Sub_Int_1
10      with Dynamic_Predicate => Sub_Int_2 /= 1;
11
12 end Custom_Integers;
```

Listing 418: show_conversion.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Custom_Integers; use Custom_Integers;
3
4 procedure Show_Conversion is
5     Int_Var      : Int;
6     Sub_Int_1_Var : Sub_Int_1;
7     Sub_Int_2_Var : Sub_Int_2;
8     Integer_Var  : Integer;
9 begin
10     Integer_Var := 5;
11     Int_Var     := Int (Integer_Var);
12
13     Put_Line ("Int_Var : "
14              & Int_Var'Image);
15
16     -- Implicit conversions:
17     -- no explicit conversion required!
18     Sub_Int_1_Var := Integer_Var;
19     Sub_Int_2_Var := Integer_Var;
20
21     Put_Line ("Sub_Int_1_Var : "
22              & Sub_Int_1_Var'Image);
23     Put_Line ("Sub_Int_2_Var : "
24              & Sub_Int_2_Var'Image);
25 end Show_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Implicit_Subtype_Conversion
 MD5: dbbe498fa66701ca94f48119b1bc1a91

Runtime output

```
Int_Var :      5
Sub_Int_1_Var : 5
Sub_Int_2_Var : 5
```

In this example, we declare the `Int` type and the `Sub_Int_1` and `Sub_Int_2` subtypes:

- the `Int` type is derived from the **Integer** type,
- `Sub_Int_1` is a subtype of the **Integer** type, and
- `Sub_Int_2` is a subtype of the `Sub_Int_1` subtype.

We need an explicit conversion when converting between the **Integer** and `Int` types. However, as the conversion is implicit for subtypes, we can simply write `Sub_Int_1_Var := Integer_Var;`. (Of course, writing the explicit conversion `Sub_Int_1 (Integer_Var)` in the assignment is possible as well.) Also, the same applies to the `Sub_Int_2` subtype: we can write an implicit conversion in the `Sub_Int_2_Var := Integer_Var;` statement.

Conversion of other types

For other kinds of types, such as records, a direct conversion as we've seen so far isn't possible. In this case, we have to write a conversion function ourselves. A common convention in Ada is to name this function `To_TypeName`. For example, if we want to convert from any type to **Integer** or **Float**, we implement the `To_Integer` and `To_Float` functions, respectively. (Obviously, because Ada supports subprogram overloading, we can have multiple `To_TypeName` functions for different operand types.)

Let's see a code example:

Listing 419: custom_rec.ads

```
1 package Custom_Rec is
2
3     type Rec is record
4         X : Integer;
5     end record;
6
7     function To_Integer (R : Rec)
8         return Integer is
9         (R.X);
10
11 end Custom_Rec;
```

Listing 420: show_conversion.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Custom_Rec; use Custom_Rec;
3
4 procedure Show_Conversion is
5     R : Rec;
6     I : Integer;
7 begin
8     R := (X => 2);
9     I := To_Integer (R);
10
```

(continues on next page)

(continued from previous page)

```

11   Put_Line ("I : " & I'Image);
12 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Other_Type_↵Conversions
 MD5: d52a4fde48243a7dd6942f0b2b91ce62

Runtime output

```
I : 2
```

In this example, we have the `To_Integer` function that converts from the `Rec` type to the **Integer** type.

In other languages

In C++, you can define conversion operators to cast between objects of different classes. Also, you can overload the `=` operator. Consider this example:

```

#include <iostream>

class T1 {
public:
    T1 (float x) :
        x(x) {}

    // If class T3 is declared before class
    // T1, we can overload the "=" operator.
    //
    // void operator=(T3 v) {
    //     x = static_cast<float>(v);
    // }

    void display();
private:
    float x;
};

class T3 {
public:
    T3 (float x, float y, float z) :
        x(x), y(y), z(z) {}

    // implicit conversion
    operator float() const {
        return (x + y + z) / 3.0;
    }

    // implicit conversion
    //
    // operator T1() const {
    //     return T1((x + y + z) / 3.0);
    // }

    // explicit conversion (C++11)
    explicit operator T1() const {
        return T1(float(*this));
    }
}

```

```

    void display();

private:
    float x, y, z;
};

void T1::display()
{
    std::cout << "(x => " << x
                << ")" << std::endl;
}

void T3::display()
{
    std::cout << "(x => " << x
                << "y => " << y
                << "z => " << z
                << ")" << std::endl;
}

int main ()
{
    const T3 t_3 (0.5, 0.4, 0.6);
    T1 t_1 (0.0);
    float f;

    // Implicit conversion
    f = t_3;

    std::cout << "f : " << f
                << std::endl;

    // Explicit conversion
    f = static_cast<float>(t_3);

    // f = (float)t_3;

    std::cout << "f : " << f
                << std::endl;

    // Explicit conversion
    t_1 = static_cast<T1>(t_3);

    // t_1 = (T1)t_3;

    std::cout << "t_1 : ";
    t_1.display();
    std::cout << std::endl;
}

```

Here, we're using **operator float()** and **operator T1()** to cast from an object of class T3 to a floating-point value and an object of class T1, respectively. (If we switch the order and declare the T3 class before the T1 class, we could overload the = operator, as you can see in the commented-out lines.)

In Ada, this kind of conversions isn't available. Instead, we have to implement conversion functions such as the To_Integer function from the previous code example. This is the corresponding implementation:

Listing 421: custom_defs.ads

```
1 package Custom_Defs is
2
3     type T1 is private;
4
5     function Init (X : Float)
6         return T1;
7
8     procedure Display (Obj : T1);
9
10    type T3 is private;
11
12    function Init (X, Y, Z : Float)
13        return T3;
14
15    function To_Float (Obj : T3)
16        return Float;
17
18    function To_T1 (Obj : T3)
19        return T1;
20
21    procedure Display (Obj : T3);
22
23 private
24     type T1 is record
25         X : Float;
26     end record;
27
28     function Init (X : Float)
29         return T1 is
30         (X => X);
31
32     type T3 is record
33         X, Y, Z : Float;
34     end record;
35
36     function Init (X, Y, Z : Float)
37         return T3 is
38         (X => X, Y => Y, Z => Z);
39
40 end Custom_Defs;
```

Listing 422: custom_defs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Defs is
4
5     procedure Display (Obj : T1) is
6     begin
7         Put_Line ("(X => "
8                 & Obj.X'Image & ")");
9     end Display;
10
11     function To_Float (Obj : T3)
12     return Float is
13     ((Obj.X + Obj.Y + Obj.Z) / 3.0);
14
15     function To_T1 (Obj : T3)
16     return T1 is
17     (Init (To_Float (Obj)));
18
19     procedure Display (Obj : T3) is
20     begin
21         Put_Line ("(X => " & Obj.X'Image
22                 & ", Y => " & Obj.Y'Image
23                 & ", Z => " & Obj.Z'Image
24                 & ")");
25     end Display;
26
27 end Custom_Defs;

```

Listing 423: show_conversion.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Custom_Defs; use Custom_Defs;
3
4 procedure Show_Conversion is
5     T_3 : constant T3 := Init (0.5, 0.4, 0.6);
6     T_1 :          T1 := Init (0.0);
7     F   : Float;
8 begin
9     -- Explicit conversion from
10    -- T3 to Float type
11    F := To_Float (T_3);
12
13    Put_Line ("F : " & F'Image);
14
15    -- Explicit conversion from
16    -- T3 to T1 type
17    T_1 := To_T1 (T_3);
18
19    Put ("T_1 : ");
20    Display (T_1);
21 end Show_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Type_Conversion.Explicit_Rec_↵
Conversion

MD5: b3e7be5488fb8026b4386063ba16aaeb

Runtime output

```

F : 5.00000E-01
T_1 : (X => 5.00000E-01)

```

In this example, we *translate* the casting operators from the C++ version by implementing the `To_Float` and `To_T1` functions. (In addition to that, we replace the C++ constructors by `Init` functions.)

25.1.10 Qualified Expressions

We already saw qualified expressions in the *Introduction to Ada* (page 91) course. As mentioned there, a qualified expression specifies the exact type or subtype that the target expression will be resolved to, and it can be either any expression in parentheses, or an aggregate:

Listing 424: `simple_integers.ads`

```
1 package Simple_Integers is
2
3   type Int is new Integer;
4
5   subtype Int_Not_Zero is Int
6     with Dynamic_Predicate => Int_Not_Zero /= 0;
7
8 end Simple_Integers;
```

Listing 425: `show_qualified_expressions.adb`

```
1 with Simple_Integers; use Simple_Integers;
2
3 procedure Show_Qualified_Expressions is
4   I : Int;
5 begin
6   -- Using qualified expression Int'(N)
7   I := Int'(0);
8 end Show_Qualified_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example
MD5: 0a83e10b51c72827e322984bd5c8009d

Here, the qualified expression `Int' (0)` indicates that the value zero is of `Int` type.

In the Ada Reference Manual

- [4.7 Qualified Expressions](#)⁵²

Verifying subtypes

Note

This feature was introduced in Ada 2022.

We can use qualified expressions to verify a subtype's predicate:

⁵² <http://www.ada-auth.org/standards/22rm/html/RM-4-7.html>

Listing 426: show_qualified_expressions.adb

```

1 with Simple_Integers; use Simple_Integers;
2
3 procedure Show_Qualified_Expressions is
4     I : Int;
5 begin
6     I := Int_Not_Zero'(0);
7 end Show_Qualified_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Qualified_Expressions.Example
 MD5: 3c4ab8ad7bf75ae029047f673aa15d70

Build output

```

show_qualified_expressions.adb:6:23: warning: expression fails predicate check on
↳ "Int_Not_Zero" [enabled by default]
show_qualified_expressions.adb:6:23: warning: check will fail at run time [-gnatw.
↳ a]

```

Runtime output

```

raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_qualified_
↳ expressions.adb:6

```

Here, the qualified expression `Int_Not_Zero'(0)` checks the dynamic predicate of the subtype. (This predicate check fails at runtime.)

25.1.11 Default initial values

In the *Introduction to Ada course* (page 65), we've seen that record components can have default values. For example:

Listing 427: defaults.ads

```

1 package Defaults is
2
3     type R is record
4         X : Positive := 1;
5         Y : Positive := 10;
6     end record;
7
8 end Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_1
 MD5: e230be602cbb24a854e71c8176c7148c

In this section, we'll extend the concept of default values to other kinds of type declarations, such as scalar types and arrays.

To assign a default value for a scalar type declaration — such as an enumeration and a new integer —, we use the `Default_Value` aspect:

Listing 428: defaults.ads

```
1 package Defaults is
2
3     type E is (E1, E2, E3)
4         with Default_Value => E1;
5
6     type T is new Integer
7         with Default_Value => -1;
8
9 end Defaults;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_2
MD5: e6cd8261b099278ceeb5fda91d318f6e

Note that we cannot specify a default value for a subtype:

Listing 429: defaults.ads

```
1 package Defaults is
2
3     subtype T is Integer
4         with Default_Value => -1;
5     -- ERROR!!
6
7 end Defaults;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_3
MD5: beef68e4a7a3714cfa3e547bdcda9a0c

Build output

```
defaults.ads:4:11: error: aspect "Default_Value" cannot apply to subtype
gprbuild: *** compilation phase failed
```

For array types, we use the Default_Component_Value aspect:

Listing 430: defaults.ads

```
1 package Defaults is
2
3     type Arr is
4         array (Positive range <>) of Integer
5         with Default_Component_Value => -1;
6
7 end Defaults;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults_4
MD5: 2c390e3900e4af42498381025a37955e

This is a package containing the declarations we've just seen:

Listing 431: defaults.ads

```
1 package Defaults is
2
```

(continues on next page)

(continued from previous page)

```

3  type E is (E1, E2, E3)
4      with Default_Value => E1;
5
6  type T is new Integer
7      with Default_Value => -1;
8
9  -- We cannot specify default
10 -- values for subtypes:
11 --
12 -- subtype T is Integer
13 --     with Default_Value => -1;
14
15 type R is record
16     X : Positive := 1;
17     Y : Positive := 10;
18 end record;
19
20 type Arr is
21     array (Positive range <>) of Integer
22     with Default_Component_Value => -1;
23
24 end Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults
MD5: e9263ff5b96523c129a3d2d9bbb5a4dd

In the example below, we declare variables of the types from the Defaults package:

Listing 432: use_defaults.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Defaults; use Defaults;
3
4  procedure Use_Defaults is
5      E1 : E;
6      T1 : T;
7      R1 : R;
8      A1 : Arr (1 .. 5);
9  begin
10     Put_Line ("Enumeration: "
11              & E'Image (E1));
12     Put_Line ("Integer type: "
13              & T'Image (T1));
14     Put_Line ("Record type: "
15              & Positive'Image (R1.X)
16              & ", "
17              & Positive'Image (R1.Y));
18
19     Put ("Array type: ");
20     for V of A1 loop
21         Put (Integer'Image (V) & " ");
22     end loop;
23     New_Line;
24 end Use_Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Default_Initial_Values.Defaults
MD5: f8e55d31cbda2447fe14eb07eaad1975

Runtime output

```
Enumeration: E1
Integer type: -1
Record type: 1, 10
Array type: -1 -1 -1 -1 -1
```

As we see in the `Use_Defaults` procedure, all variables still have their default values, since we haven't assigned any value to them.

In the Ada Reference Manual

- [3.5 Scalar Types](#)⁵³
- [3.6 Array Types](#)⁵⁴

25.1.12 Deferred Constants

Deferred constants are declarations where the value of the constant is not specified immediately, but rather *deferred* to a later point. In that sense, if a constant declaration is deferred, it is actually declared twice:

1. in the deferred constant declaration, and
2. in the full constant declaration.

The simplest form of deferred constant is the one that has a full constant declaration in the private part of the package specification. For example:

Listing 433: `deferred_constants.ads`

```
1 package Deferred_Constants is
2
3     type Speed is new Long_Float;
4
5     Light : constant Speed;
6     --      ^ deferred constant declaration
7
8 private
9
10    Light : constant Speed := 299_792_458.0;
11    --      ^ full constant declaration
12
13 end Deferred_Constants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
↳Constant_Private
MD5: f76e42326889f70fa7e1e216576f9771
```

Another form of deferred constant is the one that imports a constant from an external implementation — using the `Import` keyword. We can use this to import a constant declaration from an implementation in C. For example, we can declare the `light` constant in a C file:

⁵³ <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

⁵⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

Listing 434: constants.c

```
1 double light = 299792458.0;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
 ↳Constant_C
 MD5: 71194a329dc5adaac3e01aff143a9943

Then, we can import this constant in the Deferred_Constants package:

Listing 435: deferred_constants.ads

```
1 package Deferred_Constants is
2
3     type Speed is new Long_Float;
4
5     Light : constant Speed with
6         Import, Convention => C;
7     -- ^^^^ deferred constant
8     --      declaration; imported
9     --      from C file
10
11 end Deferred_Constants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
 ↳Constant_C
 MD5: 9355d194e973c6c6540485178b2259c9

In this case, we don't have a full declaration in the Deferred_Constants package, as the Light constant is imported from the constants.c file.

As a rule, the deferred and the full declarations should match — except, of course, for the actual value that is missing in the deferred declaration. For instance, we're not allowed to use different types in both declarations. However, we may use a subtype in the full declaration — as long as it's compatible with the type that was used in the deferred declaration. For example:

Listing 436: deferred_constants.ads

```
1 package Deferred_Constants is
2
3     type Speed is new Long_Float;
4
5     subtype Positive_Speed is
6         Speed range 0.0 .. Speed'Last;
7
8     Light : constant Speed;
9     -- ^ deferred constant declaration
10
11 private
12
13     Light : constant Positive_Speed :=
14         299_792_458.0;
15     -- ^ full constant declaration
16     --      using a subtype
17
18 end Deferred_Constants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
 ↳Constant_Subtype
 MD5: ad6e13e30bacb6d97ccfa6c7345ffb67

Here, we're using the `Speed` type in the deferred declaration of the `Light` constant, but we're using the `Positive_Speed` subtype in the full declaration.

A useful application of deferred constants is when the value of the constant is calculated using entities not meant to be compile-time visible to clients. As such, these other entities are only visible in the private part of the package, so that's where the value of the deferred constant must be computed. For example, the full constant declaration may be computed by a call to an expression function:

Listing 437: deferred_constants.ads

```

1 package Deferred_Constants is
2
3   type Speed is new Long_Float;
4
5   Light : constant Speed;
6   --      ^ deferred constant declaration
7
8 private
9
10  function Calculate_Light return Speed is
11    (299_792_458.0);
12
13  Light : constant Speed := Calculate_Light;
14  --      ^ full constant declaration
15  --      calling a private function
16
17 end Deferred_Constants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Deferred_Constants.Deferred_
 ↳Constant_Function
 MD5: f0b1a9521af31a4b48bbd54891f1c32b

Here, we call the `Calculate_Light` function — declared in the private part of the `Deferred_Constants` package — for the full declaration of the `Light` constant.

i In the Ada Reference Manual

- 7.4 Deferred Constants⁵⁵

25.1.13 User-defined literals

i Note

This feature was introduced in Ada 2022.

Any type definition has a kind of literal associated with it. For example, integer types are associated with integer literals. Therefore, we can initialize an object of integer type with an integer literal:

⁵⁵ <http://www.ada-auth.org/standards/22rm/html/RM-7-4.html>

Listing 438: simple_integer_literal.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Integer_Literal is
4   V : Integer;
5 begin
6   V := 10;
7
8   Put_Line (Integer'Image (V));
9 end Simple_Integer_Literal;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Simple_Integer_Literal
 MD5: 9f65e7c319be2b292dc1fdf02dd7cfb4

Runtime output

10

Here, `10` is the integer literal that we use to initialize the integer variable `V`. Other examples of literals are real literals and string literals, as we'll see later.

When we declare an enumeration type, we limit the set of literals that we can use to initialize objects of that type:

Listing 439: simple_enumeration.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Enumeration is
4   type Activation_State is (Unknown, Off, On);
5
6   S : Activation_State;
7 begin
8   S := On;
9   Put_Line (Activation_State'Image (S));
10 end Simple_Enumeration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Simple_Enumeration
 MD5: 075df146fcb567817dadfdb245659773

Runtime output

ON

For objects of `Activation_State` type, such as `S`, the only possible literals that we can use are `Unknown`, `Off` and `On`. In this sense, types have a constrained set of literals that can be used for objects of that type.

User-defined literals allow us to extend this set of literals. We could, for example, extend the type declaration of `Activation_State` and allow the use of integer literals for objects of that type. In this case, we need to use the `Integer_Literal` aspect and specify a function that implements the conversion from literals to the type we're declaring. For this conversion from integer literals to the `Activation_State` type, we could specify that 0 corresponds to `Off`, 1 corresponds to `On` and other values correspond to `Unknown`. We'll see the corresponding implementation later.

These are the three kinds of literals and their corresponding aspect:

Literal	Example	Aspect
Integer	1	Integer_Literal
Real	1.0	Real_Literal
String	"On"	String_Literal

For our previous `Activation_States` type, we could declare a function `Integer_To_Activation_State` that converts integer literals to one of the enumeration literals that we've specified for the `Activation_States` type:

Listing 440: `activation_states.ads`

```
1 package Activation_States is
2
3   type Activation_State is (Unknown, Off, On)
4     with Integer_Literal =>
5         Integer_To_Activation_State;
6
7   function Integer_To_Activation_State
8     (S : String)
9     return Activation_State;
10
11 end Activation_States;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.User_Defined_Literals
MD5: 37c497105ea3a5ad67f72955911eb31a

Based on this specification, we can now use an integer literal to initialize an object `S` of `Activation_State` type:

```
S : Activation_State := 1;
```

Note that we have a string parameter in the declaration of the `Integer_To_Activation_State` function, even though the function itself is only used to convert integer literals (but not string literals) to the `Activation_State` type. It's our job to process that string parameter in the implementation of the `Integer_To_Activation_State` function and convert it to an integer value — using `Integer'Value`, for example:

Listing 441: `activation_states.adb`

```
1 package body Activation_States is
2
3   function Integer_To_Activation_State
4     (S : String)
5     return Activation_State is
6   begin
7     case Integer'Value (S) is
8       when 0    => return Off;
9       when 1    => return On;
10      when others => return Unknown;
11    end case;
12  end Integer_To_Activation_State;
13
14 end Activation_States;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.User_Defined_Literals
 MD5: c130c42ee2b91e4306c0b49bd6d5d322

Let's look at a complete example that makes use of all three kinds of literals:

Listing 442: activation_states.ads

```

1 package Activation_States is
2
3   type Activation_State is (Unknown, Off, On)
4     with String_Literal =>
5         To_Activation_State,
6         Integer_Literal =>
7             Integer_To_Activation_State,
8         Real_Literal =>
9             Real_To_Activation_State;
10
11   function To_Activation_State
12     (S : Wide_Wide_String)
13     return Activation_State;
14
15   function Integer_To_Activation_State
16     (S : String)
17     return Activation_State;
18
19   function Real_To_Activation_State
20     (S : String)
21     return Activation_State;
22
23 end Activation_States;
```

Listing 443: activation_states.adb

```

1 package body Activation_States is
2
3   function To_Activation_State
4     (S : Wide_Wide_String)
5     return Activation_State
6   is
7   begin
8     if S = "Off" then
9       return Off;
10    elsif S = "On" then
11      return On;
12    else
13      return Unknown;
14    end if;
15  end To_Activation_State;
16
17  function Integer_To_Activation_State
18    (S : String)
19    return Activation_State
20  is
21  begin
22    case Integer'Value (S) is
23      when 0    => return Off;
24      when 1    => return On;
25      when others => return Unknown;
26    end case;
27  end Integer_To_Activation_State;
```

(continues on next page)

(continued from previous page)

```

28
29   function Real_To_Activation_State
30     (S : String)
31     return Activation_State
32   is
33     V : constant Float := Float'Value (S);
34   begin
35     if V < 0.0 then
36       return Unknown;
37     elsif V < 1.0 then
38       return Off;
39     else
40       return On;
41     end if;
42   end Real_To_Activation_State;
43
44 end Activation_States;

```

Listing 444: activation_examples.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Activation_States; use Activation_States;
3
4  procedure Activation_Examples is
5    S : Activation_State;
6  begin
7    S := "Off";
8    Put_Line ("String: Off => "
9              & Activation_State'Image (S));
10
11    S := 1;
12    Put_Line ("Integer: 1  => "
13              & Activation_State'Image (S));
14
15    S := 1.5;
16    Put_Line ("Real:     1.5 => "
17              & Activation_State'Image (S));
18 end Activation_Examples;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_States
MD5: 8b34393e9b624cb5620ca77d5502ec8e

Runtime output

```

String: Off  => OFF
Integer: 1   => ON
Real:      1.5 => ON

```

In this example, we're extending the declaration of the `Activation_State` type to include string and real literals. For string literals, we use the `To_Activation_State` function, which converts:

- the `"Off"` string to `Off`,
- the `"On"` string to `On`, and
- any other string to `Unknown`.

For real literals, we use the `Real_To_Activation_State` function, which converts:

- any negative number to Unknown,
- a value in the interval [0, 1) to Off, and
- a value equal or above 1.0 to On.

Note that the string parameter of To_Activation_State function — which converts string literals — is of Wide_Wide_String type, and not of **String** type, as it's the case for the other conversion functions.

In the Activation_Examples procedure, we show how we can initialize an object of Activation_State type with all kinds of literals (string, integer and real literals).

With the definition of the Activation_State type that we've seen in the complete example, we can initialize an object of this type with an enumeration literal or a string, as both forms are defined in the type specification:

Listing 445: using_string_literal.adb

```

1 with Ada.Text_IO;          use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Using_String_Literal is
5     S1 : constant Activation_State := On;
6     S2 : constant Activation_State := "On";
7 begin
8     Put_Line (Activation_State'Image (S1));
9     Put_Line (Activation_State'Image (S2));
10 end Using_String_Literal;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_States
MD5: 975a3c56e7a938a89a617dc59c5302a7

Runtime output

```

ON
ON
```

Note we need to be very careful when designing conversion functions. For example, the use of string literals may limit the kind of checks that we can do. Consider the following misspelling of the Off literal:

Listing 446: misspelling_example.adb

```

1 with Ada.Text_IO;          use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Misspelling_Example is
5     S : constant Activation_State :=
6         Offf;
7     -- ^ Error: Off is misspelled.
8 begin
9     Put_Line (Activation_State'Image (S));
10 end Misspelling_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_States
MD5: 81a8ff17e0fb8c7dce18780d8c11a6ad

Build output

```
misspelling_example.adb:6:10: error: "Offf" is undefined
misspelling_example.adb:6:10: error: possible misspelling of "Off"
gprbuild: *** compilation phase failed
```

As expected, the compiler detects this error. However, this error is accepted when using the corresponding string literal:

Listing 447: misspelling_example.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Activation_States; use Activation_States;
3
4 procedure Misspelling_Example is
5   S : constant Activation_State :=
6     "Offf";
7   -- ^ Error: Off is misspelled.
8 begin
9   Put_Line (Activation_State'Image (S));
10 end Misspelling_Example;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Activation_
States
MD5: 23b708a133ce1e24c0e9f7f72fa2eb29
```

Runtime output

UNKNOWN

Here, our implementation of `To_Activation_State` simply returns `Unknown`. In some cases, this might be exactly the behavior that we want. However, let's assume that we'd prefer better error handling instead. In this case, we could change the implementation of `To_Activation_State` to check all literals that we want to allow, and indicate an error otherwise — by raising an exception, for example. Alternatively, we could specify this in the preconditions of the conversion function:

```
function To_Activation_State
(S : Wide_Wide_String)
return Activation_State
with Pre => S = "Off" or
           S = "On" or
           S = "Unknown";
```

In this case, the precondition explicitly indicates which string literals are allowed for the `To_Activation_State` type.

User-defined literals can also be used for more complex types, such as records. For example:

Listing 448: silly_records.ads

```
1 package Silly_Records is
2
3   type Silly is record
4     X : Integer;
5     Y : Float;
6   end record
7   with String_Literal => To_Silly;
8
```

(continues on next page)

(continued from previous page)

```

9     function To_Silly (S : Wide_Wide_String)
10         return Silly;
11 end Silly_Records;

```

Listing 449: silly_records.adb

```

1 package body Silly_Records is
2
3     function To_Silly (S : Wide_Wide_String)
4         return Silly
5     is
6     begin
7         if S = "Magic" then
8             return (X => 42, Y => 42.0);
9         else
10            return (X => 0, Y => 0.0);
11        end if;
12    end To_Silly;
13
14 end Silly_Records;

```

Listing 450: silly_magic.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Silly_Records; use Silly_Records;
3
4 procedure Silly_Magic is
5     R1 : Silly;
6 begin
7     R1 := "Magic";
8     Put_Line (R1.X'Image & ", " & R1.Y'Image);
9 end Silly_Magic;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.User_Defined_Literals.Record_Literals
 MD5: 7395145a41de38dbbee117e27aec8c64

Runtime output

```
42, 4.20000E+01
```

In this example, when we initialize an object of Silly type with a string, its components are:

- set to 42 when using the "Magic" string; or
- simply set to zero when using any other string.

Obviously, this example isn't particularly useful. However, the goal is to show that this approach is useful for more complex types where a string literal (or a numeric literal) might simplify handling those types. User-defined literals let you design types in ways that, otherwise, would only be possible when using a preprocessor or a domain-specific language.

** In the Ada Reference Manual**

- [4.2.1 User-Defined Literals](#)⁵⁶

⁵⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html>

25.2 Types and Representation

25.2.1 Enumeration Representation Clauses

We have talked about the internal code of an enumeration *in another section* (page 300). We may change this internal code by using a representation clause, which has the following format:

```
for Primary_Color is (Red    => 1,
                      Green => 5,
                      Blue   => 1000);
```

The value of each code in a representation clause must be distinct. However, as you can see above, we don't need to use sequential values — the values must, however, increase for each enumeration.

We can rewrite the previous example using a representation clause:

Listing 451: days.ads

```
1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     for Day use (Mon => 2#00000001#,
8                 Tue => 2#00000010#,
9                 Wed => 2#00000100#,
10                Thu => 2#00001000#,
11                Fri => 2#00010000#,
12                Sat => 2#00100000#,
13                Sun => 2#01000000#);
14
15 end Days;
```

Listing 452: show_days.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Days;        use Days;
3
4 procedure Show_Days is
5 begin
6     for D in Day loop
7         Put_Line (Day'Image (D)
8                 & " position = "
9                 & Integer'Image (Day'Pos (D)));
10        Put_Line (Day'Image (D)
11                & " internal code = "
12                & Integer'Image
13                (Day'Enum_Rep (D)));
14    end loop;
15 end Show_Days;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Enumeration_Representation_Clauses.Enumeration_Values
 MD5: a70c3f8a967c355a4bf8f2d669f9c541

Runtime output

```

MON position      = 0
MON internal code = 1
TUE position      = 1
TUE internal code = 2
WED position      = 2
WED internal code = 4
THU position      = 3
THU internal code = 8
FRI position      = 4
FRI internal code = 16
SAT position      = 5
SAT internal code = 32
SUN position      = 6
SUN internal code = 64

```

Now, the value of the internal code is the one that we've specified in the representation clause instead of being equivalent to the value of the enumeration position.

In the example above, we're using binary values for each enumeration — basically viewing the integer value as a bit-field and assigning one bit for each enumeration. As long as we maintain an increasing order, we can use totally arbitrary values as well. For example:

Listing 453: days.ads

```

1 package Days is
2
3     type Day is (Mon, Tue, Wed,
4                 Thu, Fri,
5                 Sat, Sun);
6
7     for Day use (Mon => 5,
8                 Tue => 9,
9                 Wed => 42,
10                Thu => 49,
11                Fri => 50,
12                Sat => 66,
13                Sun => 99);
14
15 end Days;

```

25.2.2 Data Representation

The following sections provide a glimpse on attributes and aspects used for data representation. They are usually used for embedded applications because of strict requirements that are often found there. Therefore, unless you have very specific requirements for your application, in most cases, you won't need them. However, you should at least have a rudimentary understanding of them. To read a thorough overview on this topic, please refer to the *Introduction to Embedded Systems Programming* (page 1631) course.

In the Ada Reference Manual

- 13.2 Packed Types⁵⁷
- 13.3 Operational and Representation Attributes⁵⁸
- 13.5.3 Bit Ordering⁵⁹

⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-2.html>

⁵⁸ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁵⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-3.html>

25.2.3 Sizes

Ada offers multiple attributes to retrieve the size of a type or an object:

Attribute	Description
Size	Size of the representation of a subtype or an object (in bits).
Object_Size	Size of a component or an aliased object (in bits).
Component_Size	Size of a component of an array (in bits).
Storage_Size	Number of storage elements reserved for an access type or a task object.

For the first three attributes, the size is measured in bits. In the case of `Storage_Size`, the size is measured in storage elements. Note that the size information depends your target architecture. We'll discuss some examples to better understand the differences among those attributes.

Important

A storage element is the smallest element we can use to store data in memory. As we'll see soon, a storage element corresponds to a byte in many architectures.

The size of a storage element is represented by the `System.Storage_Unit` constant. In other words, the storage unit corresponds to the number of bits used for a single storage element.

In typical architectures, `System.Storage_Unit` is 8 bits. In this specific case, a storage element is equal to a byte in memory. Note, however, that `System.Storage_Unit` might have a value different than eight in certain architectures.

Size aspect

Before we discuss the size attributes, however, we briefly look into the `Size` aspect.

When we define a type, the compiler selects the most appropriate size for that data type. For example:

Listing 454: `custom_types.ads`

```

1 package Custom_Types is
2
3     type My_Int is range 0 .. 127;
4
5 end Custom_Types;
```

Listing 455: `show_sizes.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Sizes is
6 begin
7     Put_Line ("Size of My_Int: "
8               & My_Int'Size'Image
9               & " bits");
10 end Show_Sizes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↪Size_Aspect
 MD5: ddd2556d3c928bfaa29c7e3d2724a3ce

Runtime output

Size of My_Int: 7 bits

On a typical desktop PC, the compiler selects a size of 7 bits for the custom integer type `My_Int`. (In this example, we use the [Size attribute](#) (page 356), which we discuss soon.) This means that, in order to represent objects of `My_Int` type, the compiler has to reserve **at least** 7 bits. (In other words, this is the minimum requirement for that data type. We revisit this topic later on in this section.)

Depending on the requirements of your target system, however, you might have to request the compiler to select a specific size for your data type. You can do this by using the `Size` aspect. For example:

Listing 456: custom_types.ads

```
1 package Custom_Types is
2
3     type My_Int is range 0 .. 127
4       with Size => 24;
5
6 end Custom_Types;
```

Listing 457: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Sizes is
6 begin
7     Put_Line ("Size of My_Int: "
8               & My_Int'Size'Image
9               & " bits");
10 end Show_Sizes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↪Size_Aspect
 MD5: 305e345539ad159d90dd8fd55e1d16bf

Runtime output

Size of My_Int: 24 bits

The `size` aspect is a request to the compiler to verify that the expected size can be used on the target platform. You can think of this attribute as a dialog between the developer and the compiler:

(Developer) "I think that `My_Int` should be stored using at least 24 bits. Do you agree?"

(Ada compiler) "For the target platform that you selected, I can confirm that this is indeed the case."

Depending on the target platform, however, the conversation might play out like this:

(Developer) "I think that My_Int should be stored using at least 24 bits. Do you agree?"

(Ada compiler) "For the target platform that you selected, I cannot possibly do it!
COMPILATION ERROR!"

Size attributes

The Size attribute is a function that returns the minimum number of bits necessary to represent objects of that type. Remember that this number is selected by the compiler — unless, of course, we use the Size aspect as a request to the compiler to verify that the expected size can be used on the target platform.

Let's start with a code example using the Size attribute:

Listing 458: custom_types.ads

```
1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_S24 is range 0 .. 127
6         with Size => 24;
7
8 end Custom_Types;
```

Listing 459: show_sizes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Sizes is
6     V1 : UInt_7;
7     V2 : UInt_7_S24;
8 begin
9     Put_Line ("UInt_7'Size:      "
10             & UInt_7'Size'Image);
11     Put_Line ("UInt_7'Object_Size:  "
12             & UInt_7'Object_Size'Image);
13     Put_Line ("V1'Size:      "
14             & V1'Size'Image);
15     New_Line;
16
17     Put_Line ("UInt_7_S24'Size:      "
18             & UInt_7_S24'Size'Image);
19     Put_Line ("UInt_7_S24'Object_Size:  "
20             & UInt_7_S24'Object_Size'Image);
21     Put_Line ("V2'Size:      "
22             & V2'Size'Image);
23 end Show_Sizes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↳ Sizes
MD5: 6375682e32872f0865f4e6c5dc09bba3

Build output

```
show_sizes.adb:6:04: warning: variable "V1" is read but never assigned [-gnatwv]
show_sizes.adb:7:04: warning: variable "V2" is read but never assigned [-gnatwv]
```

Runtime output

```

UInt_7'Size:      7
UInt_7'Object_Size: 8
V1'Size:          8

UInt_7_S24'Size:   24
UInt_7_S24'Object_Size: 32
V2'Size:          32

```

Depending on your target architecture, you may see this output:

```

UInt_7'Size:      7
UInt_7'Object_Size: 8
V1'Size:          8

UInt_7_S24'Size:   24
UInt_7_S24'Object_Size: 32
V2'Size:          32

```

As we said above, when we use the `Size` attribute for a type `T`, we're retrieving the minimum number of bits necessary to represent objects of that type. Note that this is not the same as the actual size of an object of type `T` because the compiler will select an object size that is appropriate for the target architecture.

In the example above, the size of the `UInt_7` is 7 bits, while the most appropriate size to store objects of this type in the memory of our target architecture is 8 bits. To be more specific, the range of `UInt_7` (0 .. 127) can be perfectly represented in 7 bits. However, most target architectures don't offer 7-bit registers or 7-bit memory storage, so 8 bits is the most appropriate size in this case.

We can retrieve the size of an object of type `T` by using the `Object_Size`. Alternatively, we can use the `Size` attribute directly on objects of type `T` to retrieve their actual size — in our example, we write `V1'Size` to retrieve the size of `V1`.

Similarly, for the `UInt_7_S24` type, the `Size` attribute tells us that the type requires a minimum number of 24 bits to represent objects of that type, while the actual storage makes use of 32 bits. Note that, in this case, we've used the `Size` aspect (`with Size => 24`) to request 24 bits for the size of the `UInt_7_S24` type.

Component size

Let's continue our discussion on sizes with an example that makes use of the `Component_Size` attribute:

Listing 460: custom_types.ads

```

1 package Custom_Types is
2
3   type UInt_7 is range 0 .. 127;
4
5   type UInt_7_Array is
6     array (Positive range <>) of UInt_7;
7
8   type UInt_7_Array_Comp_32 is
9     array (Positive range <>) of UInt_7
10     with Component_Size => 32;
11
12 end Custom_Types;

```


Listing 461: show_sizes.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Custom_Types; use Custom_Types;
4
5  procedure Show_Sizes is
6      Arr_1 : UInt_7_Array (1 .. 20);
7      Arr_2 : UInt_7_Array_Comp_32 (1 .. 20);
8  begin
9      Put_Line
10         ("UInt_7_Array'Size:           "
11          & UInt_7_Array'Size'Image);
12      Put_Line
13         ("UInt_7_Array'Object_Size:      "
14          & UInt_7_Array'Object_Size'Image);
15      Put_Line
16         ("UInt_7_Array'Component_Size:    "
17          & UInt_7_Array'Component_Size'Image);
18      Put_Line
19         ("Arr_1'Component_Size:          "
20          & Arr_1'Component_Size'Image);
21      Put_Line
22         ("Arr_1'Size:                    "
23          & Arr_1'Size'Image);
24      New_Line;
25
26      Put_Line
27         ("UInt_7_Array_Comp_32'Object_Size:  "
28          & UInt_7_Array_Comp_32'Object_Size'Image);
29      Put_Line
30         ("UInt_7_Array_Comp_32'Object_Size:  "
31          & UInt_7_Array_Comp_32'Object_Size'Image);
32      Put_Line
33         ("UInt_7_Array_Comp_32'Component_Size: "
34          &
35          UInt_7_Array_Comp_32'Component_Size'Image);
36      Put_Line
37         ("Arr_2'Component_Size:            "
38          & Arr_2'Component_Size'Image);
39      Put_Line
40         ("Arr_2'Size:                      "
41          & Arr_2'Size'Image);
42      New_Line;
43  end Show_Sizes;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↳ Sizes
 MD5: e316bcb827e014075dfbf044935827ae

Build output

```

show_sizes.adb:6:04: warning: variable "Arr_1" is read but never assigned [-gnatwv]
show_sizes.adb:7:04: warning: variable "Arr_2" is read but never assigned [-gnatwv]

```

Runtime output

```

UInt_7_Array'Size:           17179869176
UInt_7_Array'Object_Size:    17179869176

```

(continues on next page)

(continued from previous page)

```

UInt_7_Array'Component_Size:      8
Arr_1'Component_Size:            8
Arr_1'Size:                      160

UInt_7_Array_Comp_32'Object_Size: 68719476704
UInt_7_Array_Comp_32'Object_Size: 68719476704
UInt_7_Array_Comp_32'Component_Size: 32
Arr_2'Component_Size:            32
Arr_2'Size:                      640

```

Depending on your target architecture, you may see this output:

```

UInt_7_Array'Size:                17179869176
UInt_7_Array'Object_Size:         17179869176
UInt_7_Array'Component_Size:      8
Arr_1'Component_Size:            8
Arr_1'Size:                      160

UInt_7_Array_Comp_32'Size:        68719476704
UInt_7_Array_Comp_32'Object_Size: 68719476704
UInt_7_Array_Comp_32'Component_Size: 32
Arr_2'Component_Size:            32
Arr_2'Size:                      640

```

Here, the value we get for `Component_Size` of the `UInt_7_Array` type is 8 bits, which matches the `UInt_7'Object_Size` — as we've seen in the previous subsection. In general, we expect the component size to match the object size of the underlying type.

However, we might have component sizes that aren't equal to the object size of the component's type. For example, in the declaration of the `UInt_7_Array_Comp_32` type, we're using the `Component_Size` aspect to query whether the size of each component can be 32 bits:

```

type UInt_7_Array_Comp_32 is
  array (Positive range <>) of UInt_7
  with Component_Size => 32;

```

If the code compiles, we see this value when we use the `Component_Size` attribute. In this case, even though `UInt_7'Object_Size` is 8 bits, the component size of the array type (`UInt_7_Array_Comp_32'Component_Size`) is 32 bits.

Note that we can use the `Component_Size` attribute with data types, as well as with actual objects of that data type. Therefore, we can write `UInt_7_Array'Component_Size` and `Arr_1'Component_Size`, for example.

This big number (17,179,869,176 bits) for `UInt_7_Array'Size` and `UInt_7_Array'Object_Size` might be surprising for you. This is due to the fact that Ada is reporting the size of the `UInt_7_Array` type for the case when the complete range is used. Considering that we specified a positive range in the declaration of the `UInt_7_Array` type, the maximum length on this machine is $2^{31} - 1$. The object size of an array type is calculated by multiplying the maximum length by the component size. Therefore, the object size of the `UInt_7_Array` type corresponds to the multiplication of $2^{31} - 1$ components (maximum length) by 8 bits (component size).

Storage size

To complete our discussion on sizes, let's look at this example of storage sizes:

Listing 462: custom_types.ads

```

1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Access is access UInt_7;
6
7 end Custom_Types;

```

Listing 463: show_sizes.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with Custom_Types; use Custom_Types;
5
6 procedure Show_Sizes is
7     AV1, AV2 : UInt_7_Access;
8 begin
9     Put_Line
10        ("UInt_7_Access'Storage_Size:      "
11         & UInt_7_Access'Storage_Size'Image);
12     Put_Line
13        ("UInt_7_Access'Storage_Size (bits):  "
14         & Integer'Image (UInt_7_Access'Storage_Size
15                          * System.Storage_Unit));
16
17     Put_Line
18        ("UInt_7'Size:      "
19         & UInt_7'Size'Image);
20     Put_Line
21        ("UInt_7_Access'Size:      "
22         & UInt_7_Access'Size'Image);
23     Put_Line
24        ("UInt_7_Access'Object_Size: "
25         & UInt_7_Access'Object_Size'Image);
26     Put_Line
27        ("AV1'Size:      "
28         & AV1'Size'Image);
29     New_Line;
30
31     Put_Line ("Allocating AV1...");
32     AV1 := new UInt_7;
33     Put_Line ("Allocating AV2...");
34     AV2 := new UInt_7;
35     New_Line;
36
37     Put_Line
38        ("AV1.all'Size:      "
39         & AV1.all'Size'Image);
40     New_Line;
41 end Show_Sizes;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↪ Sizes
 MD5: 5e652ee25b8550ac331f3ce98e24f7ba

Runtime output

```

UInt_7_Access'Storage_Size:      0
UInt_7_Access'Storage_Size (bits): 0
UInt_7'Size:                      7
UInt_7_Access'Size:              64
UInt_7_Access'Object_Size:      64
AV1'Size:                        64

Allocating AV1...
Allocating AV2...

AV1.all'Size:                    8

```

Depending on your target architecture, you may see this output:

```

UInt_7_Access'Storage_Size:      0
UInt_7_Access'Storage_Size (bits): 0

UInt_7'Size:                      7
UInt_7_Access'Size:              64
UInt_7_Access'Object_Size:      64
AV1'Size:                        64

Allocating AV1...
Allocating AV2...

AV1.all'Size:                    8

```

As we've mentioned earlier on, `Storage_Size` corresponds to the number of storage elements reserved for an access type or a task object. In this case, we see that the storage size of the `UInt_7_Access` type is zero. This is because we haven't indicated that memory should be reserved for this data type. Thus, the compiler doesn't reserve memory and simply sets the size to zero.

Because `Storage_Size` gives us the number of storage elements, we have to multiply this value by `System.Storage_Unit` to get the total storage size in bits. (In this particular example, however, the multiplication doesn't make any difference, as the number of storage elements is zero.)

Note that the size of our original data type `UInt_7` is 7 bits, while the size of its corresponding access type `UInt_7_Access` (and the access object `AV1`) is 64 bits. This is due to the fact that the access type doesn't contain an object, but rather memory information about an object. You can retrieve the size of an object allocated via `new` by first dereferencing it — in our example, we do this by writing `AV1.all'Size`.

Now, let's use the `Storage_Size` aspect to actually reserve memory for this data type:

Listing 464: `custom_types.ads`

```

1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Reserved_Access is access UInt_7
6         with Storage_Size => 8;
7
8 end Custom_Types;

```

Listing 465: `show_sizes.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;

```

(continues on next page)

(continued from previous page)

```

3
4 with Custom_Types; use Custom_Types;
5
6 procedure Show_Sizes is
7   RAV1, RAV2 : UInt_7_Reserved_Access;
8 begin
9   Put_Line
10    ("UInt_7_Reserved_Access'Storage_Size:      "
11     & UInt_7_Reserved_Access'Storage_Size'Image);
12
13   Put_Line
14    ("UInt_7_Reserved_Access'Storage_Size (bits): "
15     & Integer'Image
16     (UInt_7_Reserved_Access'Storage_Size
17      * System.Storage_Unit));
18
19   Put_Line
20    ("UInt_7_Reserved_Access'Size:      "
21     & UInt_7_Reserved_Access'Size'Image);
22   Put_Line
23    ("UInt_7_Reserved_Access'Object_Size: "
24     & UInt_7_Reserved_Access'Object_Size'Image);
25   Put_Line
26    ("RAV1'Size:      "
27     & RAV1'Size'Image);
28   New_Line;
29
30   Put_Line ("Allocating RAV1...");
31   RAV1 := new UInt_7;
32   Put_Line ("Allocating RAV2...");
33   RAV2 := new UInt_7;
34   New_Line;
35 end Show_Sizes;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↳ Sizes
 MD5: 6ac085d8467a61ba4f9cd138c024442d

Runtime output

```

UInt_7_Reserved_Access'Storage_Size:      8
UInt_7_Reserved_Access'Storage_Size (bits): 64
UInt_7_Reserved_Access'Size:      64
UInt_7_Reserved_Access'Object_Size: 64
RAV1'Size:      64

Allocating RAV1...
Allocating RAV2...

raised STORAGE_ERROR : s-poosiz.adb:108 explicit raise

```

Depending on your target architecture, you may see this output:

```

UInt_7_Reserved_Access'Storage_Size:      8
UInt_7_Reserved_Access'Storage_Size (bits): 64

UInt_7_Reserved_Access'Size:      64
UInt_7_Reserved_Access'Object_Size: 64
RAV1'Size:      64

```

(continues on next page)

(continued from previous page)

```

Allocating RAV1...
Allocating RAV2...

raised STORAGE_ERROR : s-poosiz.adb:108 explicit raise

```

In this case, we're reserving 8 storage elements in the declaration of `UInt_7_Reserved_Access`.

```

type UInt_7_Reserved_Access is access UInt_7
  with Storage_Size => 8;

```

Since each storage element corresponds to one byte (8 bits) in this architecture, we're reserving a maximum of 64 bits (or 8 bytes) for the `UInt_7_Reserved_Access` type.

This example raises an exception at runtime — a storage error, to be more specific. This is because the maximum reserved size is 64 bits, and the size of a single access object is 64 bits as well. Therefore, after the first allocation, the reserved storage space is already consumed, so we cannot allocate a second access object.

This behavior might be quite limiting in many cases. However, for certain applications where memory is very constrained, this might be exactly what we want to see. For example, having an exception being raised when the allocated memory for this data type has reached its limit might allow the application to have enough memory to at least handle the exception gracefully.

25.2.4 Alignment

For many algorithms, it's important to ensure that we're using the appropriate alignment. This can be done by using the `Alignment` attribute and the `Alignment` aspect. Let's look at this example:

Listing 466: `custom_types.ads`

```

1 package Custom_Types is
2
3   type UInt_7 is range 0 .. 127;
4
5   type Aligned_UInt_7 is new UInt_7
6     with Alignment => 4;
7
8 end Custom_Types;

```

Listing 467: `show_alignment.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Alignment is
6   V      : constant UInt_7      := 0;
7   Aligned_V : constant Aligned_UInt_7 := 0;
8 begin
9   Put_Line
10    ("UInt_7'Alignment:      "
11    & UInt_7'Alignment'Image);
12   Put_Line
13    ("UInt_7'Size:          "
14    & UInt_7'Size'Image);
15   Put_Line

```

(continues on next page)

(continued from previous page)

```

16     ("UInt_7'Object_Size:      "
17     & UInt_7'Object_Size'Image);
18 Put_Line
19     ("V'Alignment:            "
20     & V'Alignment'Image);
21 Put_Line
22     ("V'Size:                  "
23     & V'Size'Image);
24 New_Line;
25
26 Put_Line
27     ("Aligned_UInt_7'Alignment:  "
28     & Aligned_UInt_7'Alignment'Image);
29 Put_Line
30     ("Aligned_UInt_7'Size:      "
31     & Aligned_UInt_7'Size'Image);
32 Put_Line
33     ("Aligned_UInt_7'Object_Size: "
34     & Aligned_UInt_7'Object_Size'Image);
35 Put_Line
36     ("Aligned_V'Alignment:       "
37     & Aligned_V'Alignment'Image);
38 Put_Line
39     ("Aligned_V'Size:           "
40     & Aligned_V'Size'Image);
41 New_Line;
42 end Show_Alignment;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 Alignment
 MD5: a2fea340559193c293ccaee226de2558

Runtime output

```

UInt_7'Alignment:      1
UInt_7'Size:           7
UInt_7'Object_Size:    8
V'Alignment:           1
V'Size:                8

Aligned_UInt_7'Alignment:  4
Aligned_UInt_7'Size:      7
Aligned_UInt_7'Object_Size: 32
Aligned_V'Alignment:      4
Aligned_V'Size:          32

```

Depending on your target architecture, you may see this output:

```

UInt_7'Alignment:      1
UInt_7'Size:           7
UInt_7'Object_Size:    8
V'Alignment:           1
V'Size:                8

Aligned_UInt_7'Alignment:  4
Aligned_UInt_7'Size:      7
Aligned_UInt_7'Object_Size: 32
Aligned_V'Alignment:      4

```

(continues on next page)

(continued from previous page)

Aligned_V'Size: 32

In this example, we're reusing the `UInt_7` type that we've already been using in previous examples. Because we haven't specified any alignment for the `UInt_7` type, it has an alignment of 1 storage unit (or 8 bits). However, in the declaration of the `Aligned_UInt_7` type, we're using the `Alignment` aspect to request an alignment of 4 storage units (or 32 bits):

```
type Aligned_UInt_7 is new UInt_7
  with Alignment => 4;
```

When using the `Alignment` attribute for the `Aligned_UInt_7` type, we can confirm that its alignment is indeed 4 storage units (bytes).

Note that we can use the `Alignment` attribute for both data types and objects — in the code above, we're using `UInt_7'Alignment` and `V'Alignment`, for example.

Because of the alignment we're specifying for the `Aligned_UInt_7` type, its size — indicated by the `Object_Size` attribute — is 32 bits instead of 8 bits as for the `UInt_7` type.

Note that you can also retrieve the alignment associated with a class using `S'Class'Alignment`. For example:

Listing 468: `show_class_alignment.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Class_Alignment is
4
5   type Point_1D is tagged record
6     X : Integer;
7   end record;
8
9   type Point_2D is new Point_1D with record
10    Y : Integer;
11  end record
12    with Alignment => 16;
13
14  type Point_3D is new Point_2D with record
15    Z : Integer;
16  end record;
17
18 begin
19   Put_Line ("1D_Point'Alignment: "
20     & Point_1D'Alignment'Image);
21   Put_Line ("1D_Point'Class'Alignment: "
22     & Point_1D'Class'Alignment'Image);
23   Put_Line ("2D_Point'Alignment: "
24     & Point_2D'Alignment'Image);
25   Put_Line ("2D_Point'Class'Alignment: "
26     & Point_2D'Class'Alignment'Image);
27   Put_Line ("3D_Point'Alignment: "
28     & Point_3D'Alignment'Image);
29   Put_Line ("3D_Point'Class'Alignment: "
30     & Point_3D'Class'Alignment'Image);
31 end Show_Class_Alignment;
```


25.2.5 Overlapping Storage

Algorithms can be designed to perform in-place or out-of-place processing. In other words, they can take advantage of the fact that input and output arrays share the same storage space or not.

We can use the `Has_Same_Storage` and the `Overlaps_Storage` attributes to retrieve more information about how the storage space of two objects related to each other:

- the `Has_Same_Storage` attribute indicates whether two objects have the exact same storage.
 - A typical example is when both objects are exactly the same, so they obviously share the same storage. For example, for array A, `A'Has_Same_Storage (A)` is always **True**.
- the `Overlaps_Storage` attribute indicates whether two objects have at least one bit in common.
 - Note that, if two objects have the same storage, this implies that their storage also overlaps. In other words, `A'Has_Same_Storage (B) = True` implies that `A'Overlaps_Storage (B) = True`.

Let's look at this example:

Listing 469: `int_array_processing.ads`

```

1 package Int_Array_Processing is
2
3   type Int_Array is
4     array (Positive range <>) of Integer;
5
6   procedure Show_Storage (X : Int_Array;
7                           Y : Int_Array);
8
9   procedure Process (X : Int_Array;
10                     Y : out Int_Array);
11
12 end Int_Array_Processing;
```

Listing 470: `int_array_processing.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Int_Array_Processing is
4
5   procedure Show_Storage (X : Int_Array;
6                           Y : Int_Array) is
7   begin
8     if X'Has_Same_Storage (Y) then
9       Put_Line
10        ("Info: X and Y have the same storage.");
11     else
12       Put_Line
13        ("Info: X and Y don't have"
14         & "the same storage.");
15     end if;
16     if X'Overlaps_Storage (Y) then
17       Put_Line
18        ("Info: X and Y overlap.");
19     else
20       Put_Line
21        ("Info: X and Y don't overlap.");
22     end if;
```

(continues on next page)

(continued from previous page)

```

23  end Show_Storage;
24
25  procedure Process (X :      Int_Array;
26                    Y : out Int_Array) is
27  begin
28      Put_Line ("==== PROCESS ====");
29      Show_Storage (X, Y);
30
31      if X'Has_Same_Storage (Y) then
32          Put_Line ("In-place processing...");
33      else
34          if not X'Overlaps_Storage (Y) then
35              Put_Line
36                  ("Out-of-place processing...");
37          else
38              Put_Line
39                  ("Cannot process "
40                  & "overlapping arrays...");
41          end if;
42      end if;
43      New_Line;
44  end Process;
45
46  end Int_Array_Processing;

```

Listing 471: main.adb

```

1  with Int_Array_Processing;
2  use  Int_Array_Processing;
3
4  procedure Main is
5      A : Int_Array (1 .. 20) := (others => 3);
6      B : Int_Array (1 .. 20) := (others => 4);
7  begin
8      Process (A, A);
9      -- In-place processing:
10     -- sharing the exact same storage
11
12     Process (A (1 .. 10), A (10 .. 20));
13     -- Overlapping one component: A (10)
14
15     Process (A (1 .. 10), A (11 .. 20));
16     -- Out-of-place processing:
17     -- same array, but not sharing any storage
18
19     Process (A, B);
20     -- Out-of-place processing:
21     -- two different arrays
22 end Main;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↳ Overlapping_Storage
MD5: 0f599163c6f24c3ef46ec6577b501c21

Build output

int_array_processing.adb:29:24: warning: "Y" may be referenced before it has a
↳ value [enabled by default]

Runtime output

```
==== PROCESS ====
Info: X and Y have the same storage.
Info: X and Y overlap.
In-place processing...

==== PROCESS ====
Info: X and Y don't havethe same storage.
Info: X and Y overlap.
Cannot process overlapping arrays...

==== PROCESS ====
Info: X and Y don't havethe same storage.
Info: X and Y don't overlap.
Out-of-place processing...

==== PROCESS ====
Info: X and Y don't havethe same storage.
Info: X and Y don't overlap.
Out-of-place processing...
```

In this code example, we implement two procedures:

- `Show_Storage`, which shows storage information about two arrays by using the `Has_Same_Storage` and `Overlaps_Storage` attributes.
- `Process`, which are supposed to process an input array `X` and store the processed data in the output array `Y`.
 - Note that the implementation of this procedure is actually just a mock-up, so that no processing is actually taking place.

We have four different instances of how we can call the `Process` procedure:

- in the `Process (A, A)` call, we're using the same array for the input and output arrays. This is a perfect example of in-place processing. Because the input and the output arrays arguments are actually the same object, they obviously share the exact same storage.
- in the `Process (A (1 .. 10), A (10 .. 20))` call, we're using two slices of the `A` array as input and output arguments. In this case, a single component of the `A` array is shared: `A (10)`. Because the storage space is overlapping, but not exactly the same, neither in-place nor out-of-place processing can usually be used in this case.
- in the `Process (A (1 .. 10), A (11 .. 20))` call, even though we're using the same array `A` for the input and output arguments, we're using slices that are completely independent from each other, so that the input and output arrays are not sharing any storage in this case. Therefore, we can use out-of-place processing.
- in the `Process (A, B)` call, we have two different arrays — which obviously don't share any storage space —, so we can use out-of-place processing.

25.2.6 Packed Representation

As we've seen previously, the minimum number of bits required to represent a data type might be less than the actual number of bits used to store an object of that same type. We've seen an example where `UInt_7'Size` was 7 bits, while `UInt_7'Object_Size` was 8 bits. The most extreme case is the one for the `Boolean` type: in this case, `Boolean'Size` is 1 bit, while `Boolean'Object_Size` might be 8 bits (or even more on certain architectures). In such cases, we have 7 (or more) unused bits in memory for each object of `Boolean` type. In other words, we're wasting memory. On the other hand, we're gaining speed of access

because we can directly access each element without having to first change its internal representation back and forth. We'll come back to this point later.

The situation is even worse when implementing bit-fields, which can be declared as an array of **Boolean** components. For example:

Listing 472: flag_definitions.ads

```

1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6 end Flag_Definitions;
```

Listing 473: show_flags.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Flags is
5     Flags_1 : Flags (1 .. 8);
6 begin
7     Put_Line ("Boolean'Size:      "
8         & Boolean'Size'Image);
9     Put_Line ("Boolean'Object_Size:  "
10        & Boolean'Object_Size'Image);
11    Put_Line ("Flags_1'Size:      "
12        & Flags_1'Size'Image);
13    Put_Line ("Flags_1'Component_Size: "
14        & Flags_1'Component_Size'Image);
15 end Show_Flags;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
 ↪ Non_Packed_Flags
 MD5: 6fd7a913e3c6717e846c2e822c1cbad7

Build output

```
show_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned [-
  ↪ gnatwv]
```

Runtime output

```

Boolean'Size:      1
Boolean'Object_Size:  8
Flags_1'Size:      64
Flags_1'Component_Size: 8
```

Depending on your target architecture, you may see this output:

```

Boolean'Size:      1
Boolean'Object_Size:  8
Flags_1'Size:      64
Flags_1'Component_Size: 8
```

In this example, we're declaring the **Flags** type as an array of **Boolean** components. As we can see in this case, although the size of the **Boolean** type is just 1 bit, an object of this type has a size of 8 bits. Consequently, each component of the **Flags** type has a size of 8 bits. Moreover, an array with 8 components of **Boolean** type — such as the **Flags_1** array — has a size of 64 bits.

Therefore, having a way to compact the representation — so that we can store multiple objects without wasting storage space — may help us improving memory usage. This is actually possible by using the Pack aspect. For example, we could extend the previous example and declare a Packed_Flags type that makes use of this aspect:

Listing 474: flag_definitions.ads

```
1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6     type Packed_Flags is
7         array (Positive range <>) of Boolean
8         with Pack;
9
10 end Flag_Definitions;
```

Listing 475: show_packed_flags.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Packed_Flags is
5     Flags_1 : Flags (1 .. 8);
6     Flags_2 : Packed_Flags (1 .. 8);
7 begin
8     Put_Line ("Boolean'Size:      "
9         & Boolean'Size'Image);
10    Put_Line ("Boolean'Object_Size:  "
11        & Boolean'Object_Size'Image);
12    Put_Line ("Flags_1'Size:      "
13        & Flags_1'Size'Image);
14    Put_Line ("Flags_1'Component_Size: "
15        & Flags_1'Component_Size'Image);
16    Put_Line ("Flags_2'Size:      "
17        & Flags_2'Size'Image);
18    Put_Line ("Flags_2'Component_Size: "
19        & Flags_2'Component_Size'Image);
20 end Show_Packed_Flags;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.
↳ Packed_Flags
MD5: c71cf68dc8bc41d0df2a5e3eb61b51fd

Build output

```
show_packed_flags.adb:5:04: warning: variable "Flags_1" is read but never assigned,
↳ [-gnatwv]
show_packed_flags.adb:6:04: warning: variable "Flags_2" is read but never assigned,
↳ [-gnatwv]
```

Runtime output

```
Boolean'Size:      1
Boolean'Object_Size: 8
Flags_1'Size:      64
Flags_1'Component_Size: 8
Flags_2'Size:      8
Flags_2'Component_Size: 1
```

Depending on your target architecture, you may see this output:

```
Boolean'Size:      1
Boolean'Object_Size: 8
Flags_1'Size:      64
Flags_1'Component_Size: 8
Flags_2'Size:      8
Flags_2'Component_Size: 1
```

In this example, we're declaring the `Flags_2` array of `Packed_Flags` type. Its size is 8 bits — instead of the 64 bits required for the `Flags_1` array. Because the array type `Packed_Flags` is packed, we can now effectively use this type to store an object of `Boolean` type using just 1 bit of the memory, as indicated by the `Flags_2'Component_Size` attribute.

In many cases, we need to convert between a *normal* representation (such as the one used for the `Flags_1` array above) to a packed representation (such as the one for the `Flags_2` array). In many programming languages, this conversion may require writing custom code with manual bit-shifting and bit-masking to get the proper target representation. In Ada, however, we just need to indicate the actual type conversion, and the compiler takes care of generating code containing bit-shifting and bit-masking to performs the type conversion.

Let's modify the previous example and introduce this type conversion:

Listing 476: `flag_definitions.ads`

```
1 package Flag_Definitions is
2
3     type Flags is
4         array (Positive range <>) of Boolean;
5
6     type Packed_Flags is
7         array (Positive range <>) of Boolean
8         with Pack;
9
10    Default_Flags : constant Flags :=
11        (True, True, False, True,
12         False, False, True, True);
13
14 end Flag_Definitions;
```

Listing 477: `show_flag_conversion.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Flag_Definitions; use Flag_Definitions;
3
4 procedure Show_Flag_Conversion is
5     Flags_1 : Flags (1 .. 8);
6     Flags_2 : Packed_Flags (1 .. 8);
7 begin
8     Flags_1 := Default_Flags;
9     Flags_2 := Packed_Flags (Flags_1);
10
11     for I in Flags_2'Range loop
12         Put_Line (I'Image & ": "
13                  & Flags_1 (I)'Image & ", "
14                  & Flags_2 (I)'Image);
15     end loop;
16 end Show_Flag_Conversion;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Type_Representation.Data_Representation.`
(continues on next page)

(continued from previous page)

```
→ Flag_Conversion
MD5: faff2079f6779097b6e0f7cd6cd48612
```

Runtime output

```
1: TRUE, TRUE
2: TRUE, TRUE
3: FALSE, FALSE
4: TRUE, TRUE
5: FALSE, FALSE
6: FALSE, FALSE
7: TRUE, TRUE
8: TRUE, TRUE
```

In this extended example, we're now declaring `Default_Flags` as an array of constant flags, which we use to initialize `Flags_1`.

The actual conversion happens with `Flags_2 := Packed_Flags (Flags_1)`. Here, the type conversion `Packed_Flags ( )` indicates that we're converting from the normal representation (used for the `Flags` type) to the packed representation (used for `Packed_Flags` type). We don't need to write more code than that to perform the correct type conversion.

Also, by using the same strategy, we could read information from a packed representation. For example:

```
Flags_1 := Flags (Flags_2);
```

In this case, we use `Flags ( )` to convert from a packed representation to the normal representation.

We elaborate on the topic of converting between data representations in the section on [changing data representation](#) (page 381).

Trade-offs

As indicated previously, when we're using a packed representation (vs. using a standard *unpacked* representation), we're trading off speed of access for less memory consumption. The following table summarizes this:

Representation	More speed of access	Less memory consumption
Unpacked	X	
Packed		X

On one hand, we have better memory usage when we apply packed representations because we may save many bits for each object. On the other hand, there's a cost associated with accessing those packed objects because they need to be unpacked before we can actually access them. In fact, the compiler generates code — using bit-shifting and bit-masking — that converts a packed representation into an unpacked representation, which we can then access. Also, when storing a packed object, the compiler generates code that converts the unpacked representation of the object into the packed representation.

This packing and unpacking mechanism has a performance cost associated with it, which results in less speed of access for packed objects. As usual in those circumstances, before using packed representation, we should assess whether memory constraints are more important than speed in our target architecture.

25.2.7 Record Representation and storage clauses

In this section, we discuss how to use record representation clauses to specify how a record is represented in memory. Our goal is to provide a brief introduction into the topic. If you're interested in more details, you can find a thorough discussion about record representation clauses in the *Introduction to Embedded Systems Programming* (page 1631) course.

Let's start with the simple approach of declaring a record type without providing further information. In this case, we're basically asking the compiler to select a reasonable representation for that record in the memory of our target architecture.

Let's see a simple example:

Listing 478: p.ads

```
1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8 end P;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_1
MD5: 88171257118810bb7e02cea60ffb1ad9

Considering a typical 64-bit PC architecture with 8-bit storage units, and **Integer** defined as a 32-bit type, we get this memory representation:

position	0	1	2	3	4	5	6	7
component	A				B			

Each storage unit is a position in memory. In the graph above, the numbers on the top (0, 1, 2, ...) represent those positions for record R.

In addition, we can show the bits that are used for components A and B:

position	0	1	2	3	4	5	6	7
bits	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31
component	A				B			

The memory representation we see in the graph above can be described in Ada using representation clauses, as you can see in the code starting at the **for R use** record line in the code example below — we'll discuss the syntax and further details right after this example.

Listing 479: p.ads

```

1 package P is
2
3     type R is record
4         A : Integer;
5         B : Integer;
6     end record;
7
8     -- Representation clause for record R:
9     for R use record
10         A at 0 range 0 .. 31;
11         -- ^ starting memory position
12         B at 4 range 0 .. 31;
13         -- ^ first bit .. last bit
14     end record;
15
16 end P;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_2
MD5: b6be86ae7e1a5c2e7d981fe37bad49ed

Here, we're specifying that the A component is stored in the bits #0 up to #31 starting at position #0. Note that the position itself doesn't represent an absolute address in the device's memory; instead, it's relative to the memory space reserved for that record. The B component has the same 32-bit range, but starts at position #4.

This is a generalized view of the syntax:

```

for Record_Type use record
    Component_Name at Start_Position
                    range First_Bit .. Last_Bit;
end record;
```

These are the elements we see above:

- **Component_Name**: name of the component (from the record type declaration);
- **Start_Position**: start position — in storage units — of the memory space reserved for that component;
- **First_Bit**: first bit (in the start position) of the component;
- **Last_Bit**: last bit of the component.

Note that the last bit of a component might be in a different storage unit. Since the **Integer** type has a larger width (32 bits) than the storage unit (8 bits), components of that type span over multiple storage units. Therefore, in our example, the first bit of component A is at position #0, while the last bit is at position #31.

Also note that the last eight bits of component A are bits #24 .. #31. If we think in terms of storage units, this corresponds to bits #0 .. #7 of position #3. However, when specifying the last bit in Ada, we always use the **First_Bit** value as a reference, not the position where those bits might end up. Therefore, we write **range 0 .. 31**, well knowing that those 32 bits span over four storage units (positions #0 .. #3).

In the Ada Reference Manual

- [13.5.1 Record Representation Clauses](#)⁶⁰

Storage Place Attributes

We can retrieve information about the start position, and the first and last bits of a component by using the storage place attributes:

- `Position`, which retrieves the start position of a component;
- `First_Bit`, which retrieves the first bit of a component;
- `Last_Bit`, which retrieves the last bit of a component.

Note, however, that these attributes can only be used with actual records, and not with record types.

We can revisit the previous example and verify how the compiler represents the `R` type in memory:

Listing 480: `p.ads`

```

1 package P is
2
3     type R is record
4         A : Integer;
5         B : Integer;
6     end record;
7
8 end P;
```

Listing 481: `show_storage.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with P;          use P;
5
6 procedure Show_Storage is
7     R1 : R;
8 begin
9     Put_Line ("R'Size:           "
10              & R'Size'Image);
11     Put_Line ("R'Object_Size:    "
12              & R'Object_Size'Image);
13     New_Line;
14
15     Put_Line ("System.Storage_Unit: "
16              & System.Storage_Unit'Image);
17     New_Line;
18
19     Put_Line ("R1.A'Position  : "
20              & R1.A'Position'Image);
21     Put_Line ("R1.A'First_Bit : "
22              & R1.A'First_Bit'Image);
23     Put_Line ("R1.A'Last_Bit  : "
24              & R1.A'Last_Bit'Image);
25     New_Line;
26
27     Put_Line ("R1.B'Position  : "
28              & R1.B'Position'Image);
29     Put_Line ("R1.B'First_Bit : "
30              & R1.B'First_Bit'Image);
31     Put_Line ("R1.B'Last_Bit  : "
32              & R1.B'Last_Bit'Image);
33 end Show_Storage;
```

⁶⁰ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-1.html>

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Storage_Place_Attributes
 MD5: 05a402585ce71eb47cf972e68c02835e

Build output

show_storage.adb:7:04: warning: variable "R1" is read but never assigned [-gnatwv]

Runtime output

```
R'Size:           64
R'Object_Size:    64

System.Storage_Unit:  8

R1.A'Position  :  0
R1.A'First_Bit :  0
R1.A'Last_Bit  : 31

R1.B'Position  :  4
R1.B'First_Bit :  0
R1.B'Last_Bit  : 31
```

First of all, we see that the size of the R type is 64 bits, which can be explained by those two 32-bit integer components. Then, we see that components A and B start at positions #0 and #4, and each one makes use of bits in the range from #0 to #31. This matches the graph we've seen above.

In the Ada Reference Manual

- [13.5.2 Storage Place Attributes](#)⁶¹

Using Representation Clauses

We can use representation clauses to change the way the compiler handles memory for a record type. For example, let's say we want to have an empty storage unit between components A and B. We can use a representation clause where we specify that component B starts at position #5 instead of #4, leaving an empty byte after component A and before component B:

position	0	1	2	3	4	5	6	7	8
bits	#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31		#0 .. 7	#8 .. #15	#16 .. #23	#24 .. #31
component	A					B			

This is the code that implements that:

Listing 482: p.ads

```
1 package P is
2
3   type R is record
4     A : Integer;
```

(continues on next page)

⁶¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-5-2.html>

(continued from previous page)

```

5      B : Integer;
6  end record;
7
8  for R use record
9      A at 0 range 0 .. 31;
10     B at 5 range 0 .. 31;
11 end record;
12
13 end P;
```

Listing 483: show_empty_byte.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with P;          use P;
4
5  procedure Show_Empty_Byte is
6  begin
7      Put_Line ("R'Size:          "
8              & R'Size'Image);
9      Put_Line ("R'Object_Size: "
10             & R'Object_Size'Image);
11 end Show_Empty_Byte;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_Empty_Byte
MD5: c616e534e95a06f2e8b3052a3e8a9aab

Runtime output

```

R'Size:          72
R'Object_Size:   96
```

When running the application above, we see that, due to the extra byte in the record representation, the sizes increase. On a typical 64-bit PC, R'Size is now 72 bits, which reflects the additional eight bits that we introduced between components A and B. Depending on the target architecture, you may also see that R'Object_Size is now 96 bits, which is the size the compiler selects as the most appropriate for this record type. As we've mentioned in the previous section, we can use aspects to request a specific size to the compiler. In this case, we could use the Object_Size aspect:

Listing 484: p.ads

```

1  package P is
2
3      type R is record
4          A : Integer;
5          B : Integer;
6      end record
7      with Object_Size => 72;
8
9      for R use record
10         A at 0 range 0 .. 31;
11         B at 5 range 0 .. 31;
12     end record;
13
14 end P;
```

Listing 485: show_empty_byte.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;           use P;
4
5 procedure Show_Empty_Byte is
6 begin
7   Put_Line ("R'Size:      "
8             & R'Size'Image);
9   Put_Line ("R'Object_Size: "
10            & R'Object_Size'Image);
11 end Show_Empty_Byte;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_Empty_Byte
MD5: 9d7bae2b2aabeda4bc03752544cee9b9

Runtime output

```
R'Size:      72
R'Object_Size: 72
```

If the code compiles, R'Size and R'Object_Size should now have the same value.

Derived Types And Representation Clauses

In some cases, you might want to modify the memory representation of a record without impacting existing code. For example, you might want to use a record type that was declared in a package that you're not allowed to change. Also, you would like to modify its memory representation in your application. A nice strategy is to derive a type and use a representation clause for the derived type.

We can apply this strategy on our previous example. Let's say we would like to use record type R from package P in our application, but we're not allowed to modify package P — or the record type, for that matter. In this case, we could simply derive R as R_New and use a representation clause for R_New. This is exactly what we do in the specification of the child package P.Rep:

Listing 486: p.ads

```
1 package P is
2
3   type R is record
4     A : Integer;
5     B : Integer;
6   end record;
7
8 end P;
```

Listing 487: p-rep.ads

```
1 package P.Rep is
2
3   type R_New is new R
4     with Object_Size => 72;
5
6   for R_New use record
```

(continues on next page)

(continued from previous page)

```

7      A at 0 range 0 .. 31;
8      B at 5 range 0 .. 31;
9  end record;
10
11 end P.Rep;

```

Listing 488: show_empty_byte.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with P;          use P;
4  with P.Rep;      use P.Rep;
5
6  procedure Show_Empty_Byte is
7  begin
8      Put_Line ("R'Size:      "
9              & R'Size'Image);
10     Put_Line ("R'Object_Size: "
11             & R'Object_Size'Image);
12
13     Put_Line ("R_New'Size:    "
14             & R_New'Size'Image);
15     Put_Line ("R_New'Object_Size: "
16             & R_New'Object_Size'Image);
17 end Show_Empty_Byte;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Derived_Rep_Clauses_Empty_Byte
MD5: 3a1e0837f8bd8250f20fc7b274b869d5

Runtime output

```

R'Size:      64
R'Object_Size: 64
R_New'Size:   72
R_New'Object_Size: 72

```

When running this example, we see that the R type retains the memory representation selected by the compiler for the target architecture, while the R_New has the memory representation that we specified.

Representation on Bit Level

A very common application of representation clauses is to specify individual bits of a record. This is particularly useful, for example, when mapping registers or implementing protocols.

Let's consider the following fictitious register as an example:

bit	0	1	2	3	4	5	6	7
component	S		(reserved)		Error	V1		

Here, S is the current status, Error is a flag, and V1 contains a value. Due to the fact that we can use representation clauses to describe individual bits of a register as records, the implementation becomes as simple as this:

Listing 489: p.ads

```
1 package P is
2
3   type Status is (Ready, Waiting,
4                   Processing, Done);
5   type UInt_3 is range 0 .. 2 ** 3 - 1;
6
7   type Simple_Reg is record
8     S      : Status;
9     Error  : Boolean;
10    V1     : UInt_3;
11  end record;
12
13  for Simple_Reg use record
14    S      at 0 range 0 .. 1;
15    -- Bit #2 and 3: reserved!
16    Error  at 0 range 4 .. 4;
17    V1     at 0 range 5 .. 7;
18  end record;
19
20 end P;
```

Listing 490: show_simple_reg.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;           use P;
4
5 procedure Show_Simple_Reg is
6 begin
7   Put_Line ("Simple_Reg'Size:      "
8             & Simple_Reg'Size'Image);
9   Put_Line ("Simple_Reg'Object_Size: "
10            & Simple_Reg'Object_Size'Image);
11 end Show_Simple_Reg;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_Storage_Clauses.Rep_Clauses_Simple_Reg
MD5: cbac444336572460062f922767c226a5

Runtime output

```
Simple_Reg'Size:      8
Simple_Reg'Object_Size: 8
```

As we can see in the declaration of the Simple_Reg type, each component represents a field from our register, and it has a fixed location (which matches the register representation we see in the graph above). Any operation on the register is as simple as accessing the record component. For example:

Listing 491: show_simple_reg.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P;           use P;
```

(continues on next page)

(continued from previous page)

```

4
5 procedure Show_Simple_Reg is
6   Default : constant Simple_Reg :=
7     (S      => Ready,
8      Error => False,
9      V1     => 0);
10
11   R : Simple_Reg := Default;
12 begin
13   Put_Line ("R.S: " & R.S'Image);
14
15   R.V1 := 4;
16
17   Put_Line ("R.V1: " & R.V1'Image);
18 end Show_Simple_Reg;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Record_Representation_
 ↪Storage_Clauses.Rep_Clauses_Simple_Reg
 MD5: e442396e43d6609c1c837165bbc21641

Runtime output

```

R.S:  READY
R.V1:  4

```

As we can see in the example, to retrieve the current status of the register, we just have to write `R.S`. To update the `V1` field of the register with the value 4, we just have to write `R.V1 := 4`. No extra code — such as bit-masking or bit-shifting — is needed here.

***i* In other languages**

Some programming languages require that developers use complicated, error-prone approaches — which may include manually bit-shifting and bit-masking variables — to retrieve information from or store information to individual bits or registers. In Ada, however, this is efficiently handled by the compiler, so that developers only need to correctly describe the register mapping using representation clauses.

25.2.8 Changing Data Representation***i* Note**

This section was originally written by Robert Dewar and published as [Gem #27: Changing Data Representation](#)⁶² and [Gem #28](#)⁶³.

A powerful feature of Ada is the ability to specify the exact data layout. This is particularly important when you have an external device or program that requires a very specific format. Some examples are:

⁶² <https://www.adacore.com/gems/gem-27>

⁶³ <https://www.adacore.com/gems/gem-28>

Listing 492: communication.ads

```

1 package Communication is
2
3     type Com_Packet is record
4         Key : Boolean;
5         Id  : Character;
6         Val : Integer range 100 .. 227;
7     end record;
8
9     for Com_Packet use record
10         Key at 0 range 0 .. 0;
11         Id  at 0 range 1 .. 8;
12         Val at 0 range 9 .. 15;
13     end record;
14
15 end Communication;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Com_Packet
MD5: cbd7f5547c5b0458853ac21d03aa41f8

Build output

```

communication.ads:12:11: warning: component clause forces biased representation_
↳ for "Val" [-gnatw.b]
```

which lays out the fields of a record, and in the case of Val, forces a biased representation in which all zero bits represents 100. Another example is:

Listing 493: array_representation.ads

```

1 package Array_Representation is
2
3     type Val is (A, B, C, D, E, F, G, H);
4
5     type Arr is array (1 .. 16) of Val
6         with Component_Size => 3;
7
8 end Array_Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Array_Rep
MD5: 7eb17fc2cd415acb7c53a363fa336807

which forces the components to take only 3 bits, crossing byte boundaries as needed. A final example is:

Listing 494: enumeration_representation.ads

```

1 package Enumeration_Representation is
2
3     type Status is (Off, On, Unknown);
4     for Status use (Off      => 2#001#,
5                     On       => 2#010#,
6                     Unknown => 2#100#);
7
8 end Enumeration_Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Enum_Rep
 MD5: 3c3e9f4ae11e9bb2482588d27ba43c30

which allows specified values for an enumeration type, instead of the efficient default values of 0, 1, 2.

In all these cases, we might use these representation clauses to match external specifications, which can be very useful. The disadvantage of such layouts is that they are inefficient, and accessing individual components, or, in the case of the enumeration type, looping through the values can increase space and time requirements for the program code.

One approach that is often effective is to read or write the data in question in this specified form, but internally in the program represent the data in the normal default layout, allowing efficient access, and do all internal computations with this more efficient form.

To follow this approach, you will need to convert between the efficient format and the specified format. Ada provides a very convenient method for doing this, as described in [RM 13.6 "Change of Representation"](#)⁶⁴.

The idea is to use type derivation, where one type has the specified format and the other has the normal default format. For instance for the array case above, we would write:

Listing 495: array_representation.ads

```

1 package Array_Representation is
2
3   type Val is (A, B, C, D, E, F, G, H);
4   type Arr is array (1 .. 16) of Val;
5
6   type External_Arr is new Arr
7     with Component_Size => 3;
8
9 end Array_Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Array_Rep
 MD5: d4e90f6ef8ff81771980771356eab235

Now we read and write the data using the External_Arr type. When we want to convert to the efficient form, Arr, we simply use a type conversion.

Listing 496: using_array_for_io.adb

```

1 with Array_Representation;
2 use Array_Representation;
3
4 procedure Using_Array_For_IO is
5   Input_Data : External_Arr;
6   Work_Data  : Arr;
7   Output_Data : External_Arr;
8 begin
9   -- (read data into Input_Data)
10
11   -- Now convert to internal form
12   Work_Data := Arr (Input_Data);
13
```

(continues on next page)

⁶⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-6.html>

(continued from previous page)

```

14  -- (computations using efficient
15  --   Work_Data form)
16
17  -- Convert back to external form
18  Output_Data := External_Arr (Work_Data);
19
20  end Using_Array_For_IO;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Array_Rep
 MD5: 88efe4b8a7f07e0c32f11131d6eafbc1

Build output

```
using_array_for_io.adb:5:04: warning: variable "Input_Data" is read but never
assigned [-gnatwv]
```

Using this approach, the quite complex task of copying all the data of the array from one form to another, with all the necessary masking and shift operations, is completely automatic.

Similar code can be used in the record and enumeration type cases. It is even possible to specify two different representations for the two types, and convert from one form to the other, as in:

Listing 497: enumeration_representation.ads

```

1  package Enumeration_Representation is
2
3      type Status_In is (Off, On, Unknown);
4      type Status_Out is new Status_In;
5
6      for Status_In use (Off      => 2#001#,
7                          On       => 2#010#,
8                          Unknown => 2#100#);
9
10     for Status_Out use (Off      => 103,
11                        On       => 1045,
12                        Unknown => 7700);
13
14 end Enumeration_Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Enum_Rep
 MD5: f78c3718280f9265ff54270c5834b458

There are two restrictions that must be kept in mind when using this feature. First, you have to use a derived type. You can't put representation clauses on subtypes, which means that the conversion must always be explicit. Second, there is a rule [RM 13.1](http://www.ada-auth.org/standards/22rm/html/RM-13-1.html)⁶⁵ (10) that restricts the placement of interesting representation clauses:

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

All the representation clauses that are interesting from the point of view of change of representation are "type related", so for example, the following sequence would be illegal:

⁶⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-1.html>

Listing 498: array_representation.ads

```

1 package Array_Representation is
2
3     type Val is (A, B, C, D, E, F, G, H);
4     type Arr is array (1 .. 16) of Val;
5
6     procedure Rearrange (Arg : in out Arr);
7
8     type External_Arr is new Arr
9         with Component_Size => 3;
10
11 end Array_Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.Array_Rep_2
MD5: 70201932d40e3fb356bcd8ab188f2df

Build output

```

array_representation.ads:9:11: error: representation item not permitted before Ada_2022
array_representation.ads:9:11: error: parent type "Arr" has primitive operations
gprbuild: *** compilation phase failed
```

Why these restrictions? Well, the answer is a little complex, and has to do with efficiency considerations, which we will address below.

Restrictions

In the previous subsection, we discussed the use of derived types and representation clauses to achieve automatic change of representation. More accurately, this feature is not completely automatic, since it requires you to write an explicit conversion. In fact there is a principle behind the design here which says that a change of representation should never occur implicitly behind the back of the programmer without such an explicit request by means of a type conversion.

The reason for that is that the change of representation operation can be very expensive, since in general it can require component by component copying, changing the representation on each component.

Let's have a look at the -gnatG expanded code to see what is hidden under the covers here. For example, the conversion `Arr (Input_Data)` from the previous example generates the following expanded code:

```

B26b : declare
    [subtype p__TarrD1 is integer range 1 .. 16]
    R25b : p__TarrD1 := 1;
begin
    for L24b in 1 .. 16 loop
        [subtype p__arr__XP3 is
            system__unsigned_types__long_long_unsigned range 0 ..
            16#FFFF_FFFF_FFFF#]
        work_data := p__arr__XP3!((work_data and not shift_left!(
            16#7#, 3 * (integer(L24b - 1)))) or shift_left!(p__arr__XP3!
            (input_data (R25b)), 3 * (integer(L24b - 1))));
        R25b := p__TarrD1'succ(R25b);
    end loop;
end B26b;
```

That's pretty horrible! In fact, we could have simplified it for this section, but we have left it in its original form, so that you can see why it is nice to let the compiler generate all this stuff so you don't have to worry about it yourself.

Given that the conversion can be pretty inefficient, you don't want to convert backwards and forwards more than you have to, and the whole approach is only worthwhile if we'll be doing extensive computations involving the value.

The expense of the conversion explains two aspects of this feature that are not obvious. First, why do we require derived types instead of just allowing subtypes to have different representations, avoiding the need for an explicit conversion?

The answer is precisely that the conversions are expensive, and you don't want them happening behind your back. So if you write the explicit conversion, you get all the gobbledygook listed above, but you can be sure that this never happens unless you explicitly ask for it.

This also explains the restriction we mentioned in previous subsection from [RM 13.1](#)⁶⁶ (10):

10 For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

It turns out this restriction is all about avoiding implicit changes of representation. Let's have a look at how type derivation works when there are primitive subprograms defined at the point of derivation. Consider this example:

Listing 499: my_ints.ads

```
1 package My_Ints is
2
3     type My_Int_1 is range 1 .. 10;
4
5     function Odd (Arg : My_Int_1)
6         return Boolean;
7
8     type My_Int_2 is new My_Int_1;
9
10 end My_Ints;
```

Listing 500: my_ints.adb

```
1 package body My_Ints is
2
3     function Odd (Arg : My_Int_1)
4         return Boolean is
5         (True);
6     -- Dummy implementation!
7
8 end My_Ints;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.My_Int
MD5: a29401698307998288f02b349d04d1d2

Now when we do the type derivation, we inherit the function `Odd` for `My_Int_2`. But where does this function come from? We haven't written it explicitly, so the compiler somehow materializes this new implicit function. How does it do that?

We might think that a complete new function is created including a body in which `My_Int_2`

⁶⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-1.html>

replaces `My_Int_1`, but that would be impractical and expensive. The actual mechanism avoids the need to do this by use of implicit type conversions. Suppose after the above declarations, we write:

Listing 501: using_my_int.adb

```

1  with My_Ints; use My_Ints;
2
3  procedure Using_My_Int is
4      Var : My_Int_2;
5  begin
6
7      if Odd (Var) then
8          -- ^ Calling Odd function
9          --    for My_Int_2 type.
10         null;
11     end if;
12
13 end Using_My_Int;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.My_Int
MD5: f68272d55e68687b7102885313c7831b

Build output

using_my_int.adb:4:04: warning: variable "Var" is read but never assigned [-gnatwv]

The compiler translates this as:

Listing 502: using_my_int.adb

```

1  with My_Ints; use My_Ints;
2
3  procedure Using_My_Int is
4      Var : My_Int_2;
5  begin
6
7      if Odd (My_Int_1 (Var)) then
8          -- ^ Converting My_Int_2 to
9          --    My_Int_1 type before
10         --    calling Odd function.
11         null;
12     end if;
13
14 end Using_My_Int;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Changing_Data_Representation.My_Int
MD5: b3d0053c61412a2b985cd580b645e048

Build output

using_my_int.adb:4:04: warning: variable "Var" is read but never assigned [-gnatwv]

This implicit conversion is a nice trick, it means that we can get the effect of inheriting a new operation without actually having to create it. Furthermore, in a case like this, the type conversion generates no code, since `My_Int_1` and `My_Int_2` have the same representation.

But the whole point is that they might not have the same representation if one of them had a representation clause that made the representations different, and in this case the implicit conversion inserted by the compiler could be expensive, perhaps generating the junk we quoted above for the `Arr` case. Since we never want that to happen implicitly, there is a rule to prevent it.

The business of forbidding by-reference types (which includes all tagged types) is also driven by this consideration. If the representations are the same, it is fine to pass by reference, even in the presence of the conversion, but if there was a change of representation, it would force a copy, which would violate the by-reference requirement.

So to summarize this section, on the one hand Ada gives you a very convenient way to trigger these complex conversions between different representations. On the other hand, Ada guarantees that you never get these potentially expensive conversions happening unless you explicitly ask for them.

25.2.9 Valid Attribute

When receiving data from external sources, we're subjected to problems such as transmission errors. If not handled properly, erroneous data can lead to major issues in an application.

One of those issues originates from the fact that transmission errors might lead to invalid information stored in memory. When proper checks are active, using invalid information is detected at runtime and an exception is raised at this point, which might then be handled by the application.

Instead of relying on exception handling, however, we could instead ensure that the information we're about to use is valid. We can do this by using the `Valid` attribute. For example, if we have a variable `Var`, we can verify that the value stored in `Var` is valid by writing `Var'Valid`, which returns a `Boolean` value. Therefore, if the value of `Var` isn't valid, `Var'Valid` returns `False`, so we can have code that handles this situation before we actually make use of `Var`. In other words, instead of handling a potential exception in other parts of the application, we can proactively verify that input information is correct and avoid that an exception is raised.

In the next example, we show an application that

- generates a file containing mock-up data, and then
- reads information from this file as state values.

The mock-up data includes valid and invalid states.

Listing 503: `create_test_file.ads`

```
1 procedure Create_Test_File (File_Name : String);
```

Listing 504: `create_test_file.adb`

```
1 with Ada.Sequential_IO;
2
3 procedure Create_Test_File (File_Name : String)
4 is
5     package Integer_Sequential_IO is new
6         Ada.Sequential_IO (Integer);
7     use Integer_Sequential_IO;
8
9     F : File_Type;
10 begin
11     Create (F, Out_File, File_Name);
12     Write (F, 1);
```

(continues on next page)

(continued from previous page)

```

13   Write (F, 2);
14   Write (F, 4);
15   Write (F, 3);
16   Write (F, 2);
17   Write (F, 10);
18   Close (F);
19 end Create_Test_File;

```

Listing 505: states.ads

```

1 with Ada.Sequential_IO;
2
3 package States is
4
5     type State is (Off, On, Waiting)
6       with Size => Integer'Size;
7
8     for State use (Off      => 1,
9                   On        => 2,
10                  Waiting => 4);
11
12     package State_Sequential_IO is new
13       Ada.Sequential_IO (State);
14
15     procedure Read_Display_States
16       (File_Name : String);
17
18 end States;

```

Listing 506: states.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5     procedure Read_Display_States
6       (File_Name : String)
7     is
8         use State_Sequential_IO;
9
10        F : State_Sequential_IO.File_Type;
11        S : State;
12
13        procedure Display_State (S : State) is
14        begin
15            -- Before displaying the value,
16            -- check whether it's valid or not.
17            if S'Valid then
18                Put_Line (S'Image);
19            else
20                Put_Line ("Invalid value detected!");
21            end if;
22        end Display_State;
23
24    begin
25        Open (F, In_File, File_Name);
26
27        while not End_Of_File (F) loop
28            Read (F, S);
29            Display_State (S);

```

(continues on next page)

(continued from previous page)

```

30     end loop;
31
32     Close (F);
33     end Read_Display_States;
34
35 end States;

```

Listing 507: show_states_from_file.adb

```

1  with States;           use States;
2  with Create_Test_File;
3
4  procedure Show_States_From_File is
5      File_Name : constant String := "data.bin";
6  begin
7      Create_Test_File (File_Name);
8      Read_Display_States (File_Name);
9  end Show_States_From_File;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Valid_Attribute.Valid_States
 MD5: f7af2946ebe663932494448a0d3d3020

Runtime output

```

OFF
ON
WAITING
Invalid value detected!
ON
Invalid value detected!

```

Let's start our discussion on this example with the States package, which contains the declaration of the State type. This type is a simple enumeration containing three states: Off, On and Waiting. We're assigning specific integer values for this type by declaring an enumeration representation clause. Note that we're using the Size aspect to request that objects of this type have the same size as the **Integer** type. This becomes important later on when parsing data from the file.

In the Create_Test_File procedure, we create a file containing integer values, which is parsed later by the Read_Display_States procedure. The Create_Test_File procedure doesn't contain any reference to the State type, so we're not constrained to just writing information that is valid for this type. On the contrary, this procedure makes use of the **Integer** type, so we can write any integer value to the file. We use this strategy to write both valid and invalid values of State to the file. This allows us to simulate an environment where transmission errors occur.

We call the Read_Display_States procedure to read information from the file and display each state stored in the file. In the main loop of this procedure, we call Read to read a state from the file and store it in the S variable. We then call the nested Display_State procedure to display the actual state stored in S. The most important line of code in the Display_State procedure is the one that uses the Valid attribute:

```
if S'Valid then
```

In this line, we're verifying that the S variable contains a valid state before displaying the actual information from S. If the value stored in S isn't valid, we can handle the issue accordingly. In this case, we're simply displaying a message indicating that an invalid value was

detected. If we didn't have this check, the `Constraint_Error` exception would be raised when trying to use invalid data stored in `S` — this would happen, for example, after reading the integer value 3 from the input file.

In summary, using the `Valid` attribute is a good strategy we can employ when we know that information stored in memory might be corrupted.

i In the Ada Reference Manual

- 13.9.2 The `Valid` Attribute⁶⁷

25.2.10 Unchecked Union

We've introduced variant records back in the *Introduction to Ada course* (page 104). In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a **case**. (We talk again about *variant records* (page 498) in another chapter.)

The `State_Or_Integer` declaration in the `States` package below is an example of a variant record:

Listing 508: `states.ads`

```

1 package States is
2
3     type State is (Off, On, Waiting)
4       with Size => Integer'Size;
5
6     for State use (Off      => 1,
7                   On       => 2,
8                   Waiting  => 4);
9
10    type State_Or_Integer (Use_Enum : Boolean) is
11      record
12        case Use_Enum is
13          when False => I : Integer;
14          when True  => S : State;
15        end case;
16      end record;
17
18    procedure Display_State_Value
19      (V : State_Or_Integer);
20
21 end States;
```

Listing 509: `states.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5     procedure Display_State_Value
6       (V : State_Or_Integer)
7     is
8     begin
9       Put_Line ("State: " & V.S'Image);
10      Put_Line ("Value: " & V.I'Image);
11    end Display_State_Value;
```

(continues on next page)

⁶⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-9-2.html>

(continued from previous page)

```

12
13 end States;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_
↳Or_Integer
MD5: fa72f52a4396a2e66931ff6932c567fc

```

As mentioned in the previous course, if you try to access a component that is not valid for your record, a `Constraint_Error` exception is raised. For example, in the implementation of the `Display_State_Value` procedure, we're trying to retrieve the value of the integer component (I) of the `V` record. When calling this procedure, the `Constraint_Error` exception is raised as expected because `Use_Enum` is set to `True`, so that the I component is invalid — only the S component is valid in this case.

Listing 510: show_variant_rec_error.adb

```

1 with States; use States;
2
3 procedure Show_Variant_Rec_Error is
4   V : State_Or_Integer (Use_Enum => True);
5 begin
6   V.S := 0n;
7   Display_State_Value (V);
8 end Show_Variant_Rec_Error;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_
↳Or_Integer
MD5: b8cf215dd55bfdec6950df35c7bc19b9

```

Runtime output

```

State: 0N

raised CONSTRAINT_ERROR : states.adb:10 discriminant check failed

```

In addition to not being able to read the value of a component that isn't valid, assigning a value to a component that isn't valid also raises an exception at runtime. In this example, we cannot assign to `V.I`:

Listing 511: show_variant_rec_error.adb

```

1 with States; use States;
2
3 procedure Show_Variant_Rec_Error is
4   V : State_Or_Integer (Use_Enum => True);
5 begin
6   V.I := 4;
7   -- Error: V.I cannot be accessed because
8   --           Use_Enum is set to True.
9 end Show_Variant_Rec_Error;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.State_
↳Or_Integer
MD5: 985a84facc3d590ac767e914bea0c1d

```

Build output

```
show_variant_rec_error.adb:6:05: warning: component not present in subtype of
↳ "State_Or_Integer" defined at line 4 [enabled by default]
show_variant_rec_error.adb:6:05: warning: Constraint_Error will be raised at run_
↳ time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_variant_rec_error.adb:6 discriminant check failed
```

We may circumvent this limitation by using the `Unchecked_Union` aspect. For example, we can derive a new type from `State_Or_Integer` and use this aspect in its declaration. We do this in the declaration of the `Unchecked_State_Or_Integer` type below.

Listing 512: states.ads

```
1 package States is
2
3     type State is (Off, On, Waiting)
4       with Size => Integer'Size;
5
6     for State use (Off      => 1,
7                   On       => 2,
8                   Waiting => 4);
9
10    type State_Or_Integer (Use_Enum : Boolean) is
11    record
12        case Use_Enum is
13            when False => I : Integer;
14            when True  => S : State;
15        end case;
16    end record;
17
18    type Unchecked_State_Or_Integer
19      (Use_Enum : Boolean) is new
20      State_Or_Integer (Use_Enum)
21        with Unchecked_Union;
22
23    procedure Display_State_Value
24      (V : Unchecked_State_Or_Integer);
25
26 end States;
```

Listing 513: states.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5     procedure Display_State_Value
6       (V : Unchecked_State_Or_Integer)
7     is
8     begin
9         Put_Line ("State: " & V.S'Image);
10        Put_Line ("Value: " & V.I'Image);
11    end Display_State_Value;
12
13 end States;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.  
↳Unchecked_State_Or_Integer  
MD5: e97271a24aab23d2db450308401667ac
```

Because we now use the `Unchecked_State_Or_Integer` type for the input parameter of the `Display_State_Value` procedure, no exception is raised at runtime, as both components are now accessible. For example:

Listing 514: `show_unchecked_union.adb`

```
1 with States; use States;  
2  
3 procedure Show_Unchecked_Union is  
4   V : State_Or_Integer (Use_Enum => True);  
5 begin  
6   V.S := 0n;  
7   Display_State_Value  
8     (Unchecked_State_Or_Integer (V));  
9 end Show_Unchecked_Union;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.  
↳Unchecked_State_Or_Integer  
MD5: 331cc1ab6709ab7e0062d64c55a75a6c
```

Runtime output

```
State: 0N  
Value: 2
```

Note that, in the call to the `Display_State_Value` procedure, we first need to convert the `V` argument from the `State_Or_Integer` to the `Unchecked_State_Or_Integer` type.

Also, we can assign to any of the components of a record that has the `Unchecked_Union` aspect. In our example, we can now assign to both the `S` and the `I` components of the `V` record:

Listing 515: `show_unchecked_union.adb`

```
1 with States; use States;  
2  
3 procedure Show_Unchecked_Union is  
4   V : Unchecked_State_Or_Integer  
5     (Use_Enum => True);  
6 begin  
7   V := (Use_Enum => True, S => 0n);  
8   Display_State_Value (V);  
9  
10  V := (Use_Enum => False, I => 4);  
11  Display_State_Value (V);  
12 end Show_Unchecked_Union;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.  
↳Unchecked_State_Or_Integer  
MD5: bb472e91c5e7b7e63d6246dbcf5226a0
```

Runtime output

```
State: ON
Value: 2
State: WAITING
Value: 4
```

In the example above, we're use an aggregate in the assignments to V. By doing so, we avoid that Use_Enum is set to the *wrong* component. For example:

Listing 516: show_unchecked_union.adb

```
1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
6 begin
7   V.S := 0n;
8   Display_State_Value (V);
9
10  V.I := 4;
11  -- Error: cannot directly assign to V.I,
12  --      as Use_Enum is set to True.
13
14  Display_State_Value (V);
15 end Show_Unchecked_Union;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.
 ↳ Unchecked_State_Or_Integer
 MD5: 74ac11a3effdafd3959fface295a86da

Build output

```
show_unchecked_union.adb:10:05: warning: component not present in subtype of
↳ "Unchecked_State_Or_Integer" defined at line 4 [enabled by default]
show_unchecked_union.adb:10:05: warning: Constraint_Error will be raised at run_
↳ time [enabled by default]
```

Runtime output

```
State: ON
Value: 2

raised CONSTRAINT_ERROR : show_unchecked_union.adb:10 discriminant check failed
```

Here, even though the record has the Unchecked_Union attribute, we cannot directly assign to the I component because Use_Enum is set to **True**, so only the S is accessible. We can, however, read its value, as we do in the Display_State_Value procedure.

Be aware that, due to the fact the union is not checked, we might write invalid data to the record. In the example below, we initialize the I component with 3, which is a valid integer value, but results in an invalid value for the S component, as the value 3 cannot be mapped to the representation of the State type.

Listing 517: show_unchecked_union.adb

```
1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
```

(continues on next page)

(continued from previous page)

```

6 begin
7   V := (Use_Enum => False, I => 3);
8   Display_State_Value (V);
9 end Show_Unchecked_Union;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Unchecked_Union.
 ↳Unchecked_State_Or_Integer
 MD5: f63e64df137cfc3c29e41f784306f0e4

Runtime output

```
raised CONSTRAINT_ERROR : states.adb:9 invalid data
```

To mitigate this problem, we could use the Valid attribute — discussed in the previous section — for the S component before trying to use its value in the implementation of the Display_State_Value procedure:

Listing 518: states.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body States is
4
5   procedure Display_State_Value
6     (V : Unchecked_State_Or_Integer)
7   is
8   begin
9     if V.S'Valid then
10      Put_Line ("State: " & V.S'Image);
11    else
12      Put_Line ("State: <invalid>");
13    end if;
14    Put_Line ("Value: " & V.I'Image);
15    end Display_State_Value;
16
17 end States;

```

Listing 519: show_unchecked_union.adb

```

1 with States; use States;
2
3 procedure Show_Unchecked_Union is
4   V : Unchecked_State_Or_Integer
5     (Use_Enum => True);
6 begin
7   V := (Use_Enum => False, I => 3);
8   Display_State_Value (V);
9 end Show_Unchecked_Union;

```

However, in general, you should avoid using the Unchecked_Union aspect due to the potential issues you might introduce into your application. In the majority of the cases, you don't need it at all — except for special cases such as when interfacing with C code that makes use of union types or solving very specific problems when doing low-level programming.

- B.3.3 Unchecked Union Types⁶⁸

25.2.11 Addresses

In other languages, such as C, the concept of pointers and addresses plays a prominent role. (In fact, in C, many optimizations rely on the usage of pointer arithmetic.) The concept of addresses does exist in Ada, but it's mainly reserved for very specific applications, mostly related to low-level programming. In general, other approaches — such as using access types — are more than sufficient. (We discuss *access types* (page 881) in another chapter. Also, later on in that chapter, we discuss the *relation between access types and addresses* (page 993).) In this section, we discuss some details about using addresses in Ada.

We make use of the **Address** type, which is defined in the System package, to handle addresses. In contrast to other programming languages (such as C or C++), an address in Ada isn't an integer value: its definition depends on the compiler implementation, and it's actually driven directly by the hardware. For now, let's consider it to usually be a private type — this can be seen as an attempt to achieve application code portability, given the variations in hardware that result in different definitions of what an address actually is.

The **Address** type has support for *address comparison* (page 400) and *address arithmetic* (page 402) (also known as *pointer arithmetic* in C). We discuss these topics later in this section. First, let's talk about the **Address** attribute and the **Address** aspect.

In the Ada Reference Manual

- 13.7 The Package System⁶⁹

Address attribute

The **Address** attribute allows us to get the address of an object. For example:

Listing 520: use_address.adb

```

1 with System; use System;
2
3 procedure Use_Address is
4   I : aliased Integer := 5;
5   A : Address;
6 begin
7   A := I'Address;
8 end Use_Address;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_
 Attribute
 MD5: 1ee71b7cd3ed278647eb72f383da877f

Here, we're assigning the address of the I object to the A address.

In the GNAT toolchain

GNAT offers a very useful extension to the System package to retrieve a string for an address: `System.Address_Image`. This is the function profile:

⁶⁸ <http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html>

⁶⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>


```
function System.Address_Image
  (A : System.Address) return String;
```

We can use this function to display the address in an user message, for example:

Listing 521: show_address_attribute.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System.Address_Image;
3
4 procedure Show_Address_Attribute is
5   I : aliased Integer := 5;
6 begin
7   Put_Line ("Address : "
8             & System.Address_Image (I'Address));
9 end Show_Address_Attribute;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Show_
Address_Attribute

MD5: 72efddedc57701665594de5ee1939d3d

Runtime output

Address : 00007FFF9C6B4794

In the Ada Reference Manual

- 13.3 Operational and Representation Attributes⁷⁰
- 13.7 The Package System⁷¹

Address aspect

Usually, we let the compiler select the address of an object in memory, or let it use a register to store that object. However, we can specify the address of an object with the **Address** aspect. In this case, the compiler won't select an address automatically, but use the address that we're specifying. For example:

Listing 522: show_address.adb

```
1 with System; use System;
2 with System.Address_Image;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Address is
7
8   I_Main    : aliased Integer;
9   I_Mapped  : Integer
10              with Address => I_Main'Address;
11 begin
12   Put_Line ("I_Main'Address : "
13             & System.Address_Image
14             (I_Main'Address));
15   Put_Line ("I_Mapped'Address : "
16             & System.Address_Image
```

(continues on next page)

⁷⁰ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁷¹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

(continued from previous page)

```

17         (I_Mapped'Address));
18 end Show_Address;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_Overlay
 MD5: 6339c743b1ca2b1adf58c977540b43d5

Runtime output

```

I_Main'Address   : 00007FFD832374D4
I_Mapped'Address : 00007FFD832374D4

```

This approach allows us to create an overlay. For example:

Listing 523: simple_overlay.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Simple_Overlay is
4      type State is (Off, State_1, State_2)
5          with Size => Integer'Size;
6
7      for State use (Off      => 0,
8                     State_1 => 32,
9                     State_2 => 64);
10
11     S : State;
12     I : Integer
13     with Address => S'Address, Import, Volatile;
14 begin
15     S := State_2;
16     Put_Line ("I = " & Integer'Image (I));
17 end Simple_Overlay;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Simple_Overlay
 MD5: a65057882518824d3ea173d193a7ae67

Runtime output

```

I = 64

```

Here, I is an overlay of S, as it uses S'Address. With this approach, we can either use the enumeration directly (by using the S object of State type) or its integer representation (by using the I variable).

In the GNAT toolchain

We could call the GNAT-specific System' `To_Address` attribute when using the `Address` aspect:

Listing 524: shared_var_types.ads

```

1  with System;
2
3  package Shared_Var_Types is
4
5  private
6      R : Integer
7          with Atomic,
8          Address =>
9              System'To_Address (16#FFFF00A0#);
10
11 end Shared_Var_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Show_
 Access_Address
 MD5: 5c2d8e0a9615084c2a15f896c61adaa6

In this case, R will refer to the address in memory that we're specifying (16#FFFF00A0# in this case).

As explained in the [GNAT Reference Manual](#)⁷², the System'To_Address attribute denotes a function identical to To_Address (from the System.Storage_Elements package) except that it is a static attribute. (We talk about the *To_Address function* (page 401) function later on.)

Note that we're using the Atomic aspect here, which we discuss *in another chapter* (page 414).

i In the Ada Reference Manual

- 13.3 Operational and Representation Attributes⁷³
- 13.7 The Package System⁷⁴
- 13.7.1 The Package System.Storage_Elements⁷⁵

Address comparison

We can compare addresses using the common comparison operators. For example:

Listing 525: show_address.adb

```

1  with System; use System;
2  with System.Address_Image;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Address is
7
8      I, J : Integer;
9  begin
10     Put_Line ("I'Address : "
11              & System.Address_Image
12              (I'Address));

```

(continues on next page)

⁷² https://gcc.gnu.org/onlinedocs/gnat_rm/Attribute-To_005fAddress.html

⁷³ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁷⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

⁷⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>

(continued from previous page)

```

13 Put_Line ("J'Address : "
14           & System.Address_Image
15           (J'Address));
16
17 if I'Address = J'Address then
18   Put_Line ("I'Address = J'Address");
19 elsif I'Address < J'Address then
20   Put_Line ("I'Address < J'Address");
21 else
22   Put_Line ("I'Address > J'Address");
23 end if;
24 end Show_Address;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Address_Aspect
 MD5: 24ddb7d05159f26ef3b2ff6bcc2691e8

Runtime output

```

I'Address : 00007FFC271B281C
J'Address : 00007FFC271B2818
I'Address > J'Address

```

In this example, we compare the address of the I object with the address of the J object using the =, < and > operators.

** In the Ada Reference Manual**

- 13.7 The Package System⁷⁶

Address to integer conversion

The System.Storage_Elements package offers an integer representation of an address via the Integer_Address type, which is an integer type unrelated to common integer types such as **Integer** and **Long_Integer**. (The actual definition of Integer_Address is compiler-dependent, and it can be a signed or modular integer subtype.)

We can convert between the **Address** and Integer_Address types by using the To_Address and To_Integer functions. Let's see an example:

Listing 526: show_address.adb

```

1 with System;      use System;
2
3 with System.Storage_Elements;
4 use System.Storage_Elements;
5
6 with System.Address_Image;
7
8 with Ada.Text_IO; use Ada.Text_IO;
9
10 procedure Show_Address is
11   I      : Integer;
12   A1, A2 : Address;
13   IA     : Integer_Address;

```

(continues on next page)

⁷⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

(continued from previous page)

```
14 begin
15     A1 := I'Address;
16     IA := To_Integer (A1);
17     A2 := To_Address (IA);
18
19     Put_Line ("A1 : "
20               & System.Address_Image (A1));
21     Put_Line ("IA : "
22               & Integer_Address'Image (IA));
23     Put_Line ("A2 : "
24               & System.Address_Image (A2));
25 end Show_Address;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_Arith_Ada
MD5: 69e053886fb8e8571d6c94247dc9f30f

Runtime output

```
A1 : 00007FFD1C1B70B4
IA : 140725075013812
A2 : 00007FFD1C1B70B4
```

Here, we retrieve the address of the `I` object and store it in the `A1` address. Then, we convert `A1` to an integer address by calling `To_Integer` (and store it in `IA`). Finally, we convert this integer address back to an actual address by calling `To_Address`.

In the Ada Reference Manual

- [13.7.1 The Package `System.Storage\_Elements`](#)⁷⁷

Address arithmetic

Although Ada supports address arithmetic, which we discuss in this section, it should be reserved for very specific applications such as low-level programming. However, even in situations that require close access to the underlying hardware, using address arithmetic might not be the approach you should consider — make sure to evaluate other options first!

Ada supports address arithmetic via the `System.Storage_Elements` package, which includes operators such as `+` and `-` for addresses. Let's see a code example where we iterate over an array by incrementing an address that *points* to each component in memory:

Listing 527: `show_address.adb`

```
1 with System;      use System;
2
3 with System.Storage_Elements;
4 use System.Storage_Elements;
5
6 with System.Address_Image;
7
8 with Ada.Text_IO; use Ada.Text_IO;
9
10 procedure Show_Address is
11
```

(continues on next page)

⁷⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>

(continued from previous page)

```

12  Arr : array (1 .. 10) of Integer;
13  A   : Address := Arr'Address;
14      ^^^^^^^^^^
15  --   Initializing address object with
16  --   address of the first component of Arr.
17  --
18  --   We could write this as well:
19  --   ____ := Arr (1)'Address
20
21  begin
22      for I in Arr'Range loop
23          declare
24              Curr : Integer
25                  with Address => A;
26          begin
27              Curr := I;
28              Put_Line ("Curr'Address : "
29                      & System.Address_Image
30                      (Curr'Address));
31          end;
32
33          --
34          --   Address arithmetic
35          --
36          A := A + Storage_Offset (Integer'Size)
37                / Storage_Unit;
38          --   ~~~~~
39          --   Moving to next component
40      end loop;
41
42      for I in Arr'Range loop
43          Put_Line ("Arr ( "
44                  & Integer'Image (I)
45                  & " ) : "
46                  & Integer'Image (Arr (I)));
47      end loop;
48  end Show_Address;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_
Arith_Ada
MD5: 2c1cdd6874036fb9a527baae63a312d9

Runtime output

```

Curr'Address : 00007FFF4F7E78E0
Curr'Address : 00007FFF4F7E78E4
Curr'Address : 00007FFF4F7E78E8
Curr'Address : 00007FFF4F7E78EC
Curr'Address : 00007FFF4F7E78F0
Curr'Address : 00007FFF4F7E78F4
Curr'Address : 00007FFF4F7E78F8
Curr'Address : 00007FFF4F7E78FC
Curr'Address : 00007FFF4F7E7900
Curr'Address : 00007FFF4F7E7904
Arr ( 1 ) : 1
Arr ( 2 ) : 2
Arr ( 3 ) : 3
Arr ( 4 ) : 4
Arr ( 5 ) : 5

```

(continues on next page)

(continued from previous page)

```

Arr ( 6 ) : 6
Arr ( 7 ) : 7
Arr ( 8 ) : 8
Arr ( 9 ) : 9
Arr ( 10 ) : 10

```

In this example, we initialize the address A by retrieving the address of the first component of the array Arr. (Note that we could have written Arr(1)'Address instead of Arr'Address. In any case, the language guarantees that Arr'Address gives us the address of the first component, i.e. Arr'Address = Arr(1)'Address.)

Then, in the loop, we declare an overlay Curr using the current value of the A address. We can then operate on this overlay — here, we assign I to Curr. Finally, in the loop, we increment address A and make it *point* to the next component in the Arr array — to do so, we calculate the size of an **Integer** component in storage units. (For details on storage units, see the section on *storage size attribute* (page 359).)

In other languages

The code example above corresponds (more or less) to the following C code:

Listing 528: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int i;
6      int arr[10];
7
8      int *a = arr;
9      /* int *a = &arr[0]; */
10
11     for (i = 0; i < 10; i++)
12     {
13         *a++ = i;
14         printf("curr address: %p\n", a);
15     }
16
17     for (i = 0; i < 10; i++)
18     {
19         printf("arr[%d]: %d\n", i, arr[i]);
20     }
21
22     return 0;
23 }

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Addresses.Pointer_Arith_C
MD5: 7aa709a4d7ed6ce2346dbabc853e28c0

Runtime output

```

curr address: 0x7ffdde35a7f4
curr address: 0x7ffdde35a7f8
curr address: 0x7ffdde35a7fc
curr address: 0x7ffdde35a800
curr address: 0x7ffdde35a804
curr address: 0x7ffdde35a808
curr address: 0x7ffdde35a80c
curr address: 0x7ffdde35a810
curr address: 0x7ffdde35a814
curr address: 0x7ffdde35a818
arr[0]: 0
arr[1]: 1
arr[2]: 2
arr[3]: 3
arr[4]: 4
arr[5]: 5
arr[6]: 6
arr[7]: 7
arr[8]: 8
arr[9]: 9

```

While pointer arithmetic is very common in C, using address arithmetic in Ada is far from common, and it should be only used when it's really necessary to do so.

i In the Ada Reference Manual

- 13.3 Operational and Representation Attributes⁷⁸
- 13.7.1 The Package System.Storage_Elements⁷⁹

25.2.12 Discarding names

As we know, we can use the Image attribute of a type to get a string associated with this type. This is useful for example when we want to display a user message for an enumeration type:

Listing 529: show_enumeration_image.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Enumeration_Image is
4
5     type Months is
6         (January, February, March, April,
7          May, June, July, August, September,
8          October, November, December);
9
10    M : constant Months := January;
11 begin
12     Put_Line ("Month: "
13              & Months'Image (M));
14 end Show_Enumeration_Image;

```

Code block metadata

⁷⁸ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

⁷⁹ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-1.html>

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
↳ Enumeration_Image
MD5: 3863c5e06641d96b59edb9e76daa7560

Runtime output

Month: JANUARY

This is similar to having this code:

Listing 530: show_enumeration_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Enumeration_Image is
4
5     type Months is
6         (January, February, March, April,
7          May, June, July, August, September,
8          October, November, December);
9
10    M : constant Months := January;
11
12    function Months_Image (M : Months)
13        return String is
14    begin
15        case M is
16            when January   => return "JANUARY";
17            when February  => return "FEBRUARY";
18            when March     => return "MARCH";
19            when April     => return "APRIL";
20            when May       => return "MAY";
21            when June      => return "JUNE";
22            when July      => return "JULY";
23            when August    => return "AUGUST";
24            when September => return "SEPTEMBER";
25            when October   => return "OCTOBER";
26            when November  => return "NOVEMBER";
27            when December  => return "DECEMBER";
28        end case;
29    end Months_Image;
30
31 begin
32     Put_Line ("Month: "
33              & Months_Image (M));
34 end Show_Enumeration_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
↳ Enumeration_Image
MD5: 2db86044d2045bd9d4c3998cca36d51c

Runtime output

Month: JANUARY

Here, the Months_Image function associates a string with each month of the Months enumeration. As expected, the compiler needs to store the strings used in the Months_Image function when compiling this code. Similarly, the compiler needs to store strings for the Months enumeration for the Image attribute.

Sometimes, we don't need to call the Image attribute for a type. In this case, we could

save some storage by eliminating the strings associated with the type. Here, we can use the `Discard_Names` aspect to request the compiler to reduce — as much as possible — the amount of storage used for storing names for this type. Let's see an example:

Listing 531: `show_discard_names.adb`

```

1 procedure Show_Discard_Names is
2   pragma Warnings (Off, "is not referenced");
3
4   type Months is
5     (January, February, March, April,
6      May, June, July, August, September,
7      October, November, December)
8     with Discard_Names;
9
10  M : constant Months := January;
11  begin
12    null;
13  end Show_Discard_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Discarding_Names.
 ↳ Discard_Names
 MD5: 7891caac459a4be2096d443ca3190036

In this example, the compiler attempts to not store strings associated with the `Months` type duration compilation.

Note that the `Discard_Names` aspect is available for enumerations, exceptions, and tagged types.

In the GNAT toolchain

If we add this statement to the `Show_Discard_Names` procedure above:

```
Put_Line ("Month: "
         & Months'Image (M));
```

we see that the application displays "0" instead of "JANUARY". This is because GNAT doesn't store the strings associated with the `Months` type when we use the `Discard_Names` aspect for the `Months` type. (Therefore, the `Months'Image` attribute doesn't have that information.) Instead, the compiler uses the integer value of the enumeration, so that `Months'Image` returns the corresponding string for this integer value.

In the Ada Reference Manual

- [Aspect Discard_Names](#)⁸⁰

25.3 Shared variable control

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

⁸⁰ <http://www.ada-auth.org/standards/22rm/html/RM-C-5.html>

i In the Ada Reference Manual

- C.6 Shared Variable Control⁸¹

25.3.1 Volatile

A **volatile**⁸² object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread B may modify the value of that object between two read accesses of a thread A. Another typical example is the one of **memory-mapped I/O**⁸³, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code to store the value of that object in a register and then use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example:

Listing 532: show_volatile_object.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Object is
4   Val : Long_Float with Volatile;
5 begin
6   Val := 0.0;
7   for I in 0 .. 999 loop
8     Val := Val + 2.0 * Long_Float (I);
9   end loop;
10
11   Put_Line ("Val: " & Long_Float'Image (Val));
12 end Show_Volatile_Object;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Volatile.Object_Ada
 MD5: aa1e276e64e69813bfc3e3ef39f3dd47

Runtime output

Val: 9.990000000000000E+05

In this example, Val has the Volatile aspect, which makes the object volatile. We can also use the Volatile aspect in type declarations. For example:

Listing 533: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3   type Volatile_Long_Float is new
4     Long_Float with Volatile;
```

(continues on next page)

⁸¹ <http://www.ada-auth.org/standards/22rm/html/RM-C-6.html>

⁸² [https://en.wikipedia.org/wiki/Volatile_\(computer_programming\)](https://en.wikipedia.org/wiki/Volatile_(computer_programming))

⁸³ https://en.wikipedia.org/wiki/Memory-mapped_I/O

(continued from previous page)

```

5
6 end Shared_Var_Types;

```

Listing 534: show_volatile_type.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Shared_Var_Types; use Shared_Var_Types;
3
4 procedure Show_Volatile_Type is
5     Val : Volatile_Long_Float;
6 begin
7     Val := 0.0;
8     for I in 0 .. 999 loop
9         Val := Val + 2.0 * Volatile_Long_Float (I);
10    end loop;
11
12    Put_Line ("Val: "
13              & Volatile_Long_Float'Image (Val));
14 end Show_Volatile_Type;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Volatile.Type
MD5: 0d31156d47b2edcfb94debd016c8bb87

Runtime output

```
Val: 9.990000000000000E+05
```

Here, we're declaring a new type `Volatile_Long_Float` in the `Shared_Var_Types` package. This type is based on the `Long_Float` type and uses the `Volatile` aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the `Volatile_Components` aspect in the array declaration. For example:

Listing 535: show_volatile_array_components.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Array_Components is
4     Arr : array (1 .. 2) of Long_Float
5         with Volatile_Components;
6 begin
7     Arr := (others => 0.0);
8
9     for I in 0 .. 999 loop
10        Arr (1) := Arr (1) + 2.0 * Long_Float (I);
11        Arr (2) := Arr (2) + 10.0 * Long_Float (I);
12    end loop;
13
14    Put_Line ("Arr (1): "
15              & Long_Float'Image (Arr (1)));
16    Put_Line ("Arr (2): "
17              & Long_Float'Image (Arr (2)));
18 end Show_Volatile_Array_Components;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Volatile.Array_↵
Components

(continues on next page)

(continued from previous page)

MD5: 05b3ee20f08c5a85f5872727a61c148d

Runtime output

```
Arr (1):  9.990000000000000E+05
Arr (2):  4.995000000000000E+06
```

Note that it's possible to use the `Volatile` aspect for the array declaration as well:

Listing 536: `shared_var_types.ads`

```
1 package Shared_Var_Types is
2
3 private
4   Arr : array (1 .. 2) of Long_Float
5       with Volatile;
6
7 end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Volatile.Array
MD5: c9b7b9f94f1fac295753c7e7b9426fb2
```

Note that, if the `Volatile` aspect is specified for an object, then the `Volatile_Components` aspect is also specified automatically — if it makes sense in the context, of course. In the example above, even though `Volatile_Components` isn't specified in the declaration of the `Arr` array, it's automatically set as well.

25.3.2 Independent

When you write code to access a single object in memory, you might actually be accessing multiple objects at once. For example, when you declare types that make use of representation clauses — as we've seen in previous sections —, you might be accessing multiple objects that are grouped together in a single storage unit. For example, if you have components `A` and `B` stored in the same storage unit, you cannot update `A` without actually writing (the same value) to `B`. Those objects aren't independently addressable because, in order to access one of them, we have to actually address multiple objects at once.

When an object is independently addressable, we call it an independent object. In this case, we make sure that, when accessing that object, we won't be simultaneously accessing another object. As a consequence, this feature limits the way objects can be represented in memory, as we'll see next.

To indicate that an object is independent, we use the `Independent` aspect:

Listing 537: `shared_var_types.ads`

```
1 package Shared_Var_Types is
2
3   I : Integer with Independent;
4
5 end Shared_Var_Types;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Object
MD5: d90fef37584ca8802b8a3e3858c0095b
```

Similarly, we can use this aspect when declaring types:

Listing 538: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3     type Independent_Boolean is new Boolean
4       with Independent;
5
6     type Flags is record
7       F1 : Independent_Boolean;
8       F2 : Independent_Boolean;
9     end record;
10
11 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: 7bcbec5b73067149b14c4b1b061f803c

In this example, we're declaring the `Independent_Boolean` type and using it in the declaration of the `Flags` record type. Let's now derive the `Flags` type and use a representation clause for the derived type:

Listing 539: shared_var_types-representation.ads

```

1 package Shared_Var_Types.Representation is
2
3     type Rep_Flags is new Flags;
4
5     for Rep_Flags use record
6       F1 at 0 range 0 .. 0;
7       F2 at 0 range 1 .. 1;
8       --      ^ ERROR: start position of
9       --      ^ F2 is wrong!
10      --      ^ ERROR: F1 and F2 share the
11      --      ^ same storage unit!
12     end record;
13
14 end Shared_Var_Types.Representation;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type
MD5: bb9d5badf33401660e7e20a7cd612dab

Build output

```

shared_var_types-representation.ads:6:26: error: size for independent "F1" must be
↳multiple of Storage_Unit
shared_var_types-representation.ads:7:21: error: position for independent "F2"
↳must be multiple of Storage_Unit
shared_var_types-representation.ads:7:26: error: size for independent "F2" must be
↳multiple of Storage_Unit
gprbuild: *** compilation phase failed
```

As you can see when trying to compile this example, the representation clause that we used for `Rep_Flags` isn't following these limitations:

1. The size of each independent component must be a multiple of a storage unit.
2. The start position of each independent component must be a multiple of a storage unit.

For example, for architectures that have a storage unit of one byte — such as standard desktop computers —, this means that the size and the position of independent components must be a multiple of a byte. Let's correct the issues in the code above by:

- setting the size of each independent component to correspond to `Storage_Unit` — using a range between 0 and `Storage_Unit - 1` —, and
- setting the start position to zero.

This is the corrected version:

Listing 540: `shared_var_types-representation.ads`

```
1 with System;
2
3 package Shared_Var_Types.Representation is
4
5     type Rep_Flags is new Flags;
6
7     for Rep_Flags use record
8         F1 at 0 range 0 .. System.Storage_Unit - 1;
9         F2 at 1 range 0 .. System.Storage_Unit - 1;
10    end record;
11
12 end Shared_Var_Types.Representation;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type`
MD5: `ed57e57cd746698909a4f7ce40a29dfc`

Note that the representation that we're now using for `Rep_Flags` is most likely the representation that the compiler would have chosen for this data type. We could, however, have added an empty storage unit between `F1` and `F2` — by simply writing `F2 at 2 ...`:

Listing 541: `shared_var_types-representation.ads`

```
1 with System;
2
3 package Shared_Var_Types.Representation is
4
5     type Rep_Flags is new Flags;
6
7     for Rep_Flags use record
8         F1 at 0 range 0 .. System.Storage_Unit - 1;
9         F2 at 2 range 0 .. System.Storage_Unit - 1;
10    end record;
11
12 end Shared_Var_Types.Representation;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.Type`
MD5: `71fedf8aac7c19bca1ba3b487efa9b17`

As long as we follow the rules for independent objects, we're still allowed to use representation clauses that don't correspond to the one that the compiler might select.

For arrays, we can use the `Independent_Components` aspect:

Listing 542: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3     Flags : array (1 .. 8) of Boolean
4           with Independent_Components;
5
6 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↳ Components
 MD5: b331d0a13adf45624b664839fe4ba42c

We've just seen in a previous example that some representation clauses might not work with objects and types that have the Independent aspect. The same restrictions apply when we use the Independent_Components aspect. For example, this aspect prevents that array components are packed when the Pack aspect is used. Let's discuss the following erroneous code example:

Listing 543: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3     type Flags is
4       array (Positive range <>) of Boolean
5       with Independent_Components, Pack;
6
7     F : Flags (1 .. 8) with Size => 8;
8
9 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↳ Packed_Independent_Components
 MD5: dbaff4f2559ef8a449dad251f42cddc0

Build output

```

shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))
shared_var_types.ads:7:36: error: size for "F" too small, minimum allowed is 64
gprbuild: *** compilation phase failed
```

As expected, this code doesn't compile. Here, we can have either independent components, or packed components. We cannot have both at the same time because packed components aren't independently addressable. The compiler warns us that the Pack aspect won't have any effect on independent components. When we use the Size aspect in the declaration of F, we confirm this limitation. If we remove the Size aspect, however, the code is compiled successfully because the compiler ignores the Pack aspect and allocates a larger size for F:

Listing 544: shared_var_types.ads

```

1 package Shared_Var_Types is
2
3     type Flags is
4       array (Positive range <>) of Boolean
5       with Independent_Components, Pack;
6
7 end Shared_Var_Types;
```


Listing 545: show_flags_size.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 with Shared_Var_Types; use Shared_Var_Types;
5
6 procedure Show_Flags_Size is
7   F : Flags (1 .. 8);
8 begin
9   Put_Line ("Flags'Size:      "
10            & F'Size'Image & " bits");
11   Put_Line ("Flags (1)'Size:  "
12            & F (1)'Size'Image & " bits");
13   Put_Line ("# storage units: "
14            & Integer'Image
15              (F'Size /
16               System.Storage_Unit));
17 end Show_Flags_Size;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Independent.
 ↳Packed_Independent_Components
 MD5: b96f921b08b1d8207749517f833fc121

Build output

```

show_flags_size.adb:7:04: warning: variable "F" is read but never assigned [-
↳gnatwv]
shared_var_types.ads:5:37: warning: cannot pack independent components (RM 13.2(7))

```

Runtime output

```

Flags'Size:      64 bits
Flags (1)'Size:  8 bits
# storage units: 8

```

As you can see in the output of the application, even though we specify the Pack aspect for the Flags type, the compiler allocates eight storage units, one per each component of the F array.

25.3.3 Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

** In other contexts**

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means

that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store / load instructions, or even two 16-bit store / load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store / load instruction.

Because of this strict definition, we might have objects for which the Atomic aspect cannot be specified. Lots of machines support integer types that are larger than the native word-sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the Atomic aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:

Listing 546: shared_var_types.ads

```

1 with System;
2
3 package Shared_Var_Types is
4
5 private
6   R : Integer
7       with Atomic,
8         Address =>
9           System'To_Address (16#FFFF00A0#);
10
11 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Object
MD5: 5c2d8e0a9615084c2a15f896c61adaa6

Note that the **Address** aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register.

Later on, we talk again about the *Address aspect* (page 398) and the GNAT-specific *System'To_Address attribute* (page 399).

In addition to atomic objects, we can declare atomic types — similar to what we've seen before for volatile types. For example:

Listing 547: shared_var_types.ads

```

1 with System;
2
3 package Shared_Var_Types is
4
5   type Atomic_Integer is new Integer
6     with Atomic;
7
8 private
9   R : Atomic_Integer
10     with Address =>
11       System'To_Address (16#FFFF00A0#);
12
13 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Types
MD5: 009632ba0155d70def8281ba590f3d12

In this example, we're declaring the `Atomic_Integer` type, which is an atomic type. Objects of this type — such as `R` in this example — are automatically atomic.

We can also declare atomic array components:

Listing 548: `shared_var_types.ads`

```
1 package Shared_Var_Types is
2
3 private
4   Arr : array (1 .. 2) of Integer
5         with Atomic_Components;
6
7 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Array_
↳ Components
MD5: 7501bdf618621a822d451da8d731ef75

This example shows the declaration of the `Arr` array, which has atomic components — the atomicity of its components is indicated by the `Atomic_Components` aspect.

Note that if an object is atomic, it is also volatile and independent. In other words, these type declarations are equivalent:

Listing 549: `shared_var_types.ads`

```
1 package Shared_Var_Types is
2
3   type Atomic_Integer_1 is new Integer
4     with Atomic;
5
6   type Atomic_Integer_2 is new Integer
7     with Atomic,
8         Volatile,
9         Independent;
10
11 end Shared_Var_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic.Volatile_
↳ Independent
MD5: 3034c7a07698491f961d9b4fb74f03d8

A similar rule applies to components of an array. When we use the `Atomic_Components`, the following aspects are implied: `Volatile`, `Volatile_Components` and `Independent_Components`. For example, these array declarations are equivalent:

Listing 550: `shared_var_types.ads`

```
1 package Shared_Var_Types is
2
3   Arr_1 : array (1 .. 2) of Integer
4         with Atomic_Components;
5
```

(continues on next page)

(continued from previous page)

```

6   Arr_2 : array (1 .. 2) of Integer
7       with Atomic_Components,
8           Volatile,
9           Volatile_Components,
10          Independent_Components;
11
12 end Shared_Var_Types;

```

25.3.4 Full-access only

Note

This feature was introduced in Ada 2022.

A full-access object is an object that requires that read or write operations on this object are performed by reading or writing all bits of the object (i.e. the *full object*) at once. Accordingly, a full-access type is a type whose objects follow this requirement. Note that a full-access type must be simultaneously a *volatile type* (page 408) or an *atomic type* (page 414). (In other words, if a type is neither volatile nor atomic, it cannot be a full-access type.)

Important

Just as a reminder, any atomic type is automatically also *volatile* (page 408) and *independent* (page 410).

Let's see some examples:

Listing 551: show_full_access_only_types.ads

```

1 package Show_Full_Access_Only_Types is
2
3   type Nonatomic_Full_Access_Type is
4       new Long_Float
5       with Volatile, Full_Access_Only;
6
7   type Atomic_Full_Access_Type is
8       new Long_Float
9       with Atomic, Full_Access_Only;
10
11 end Show_Full_Access_Only_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
Control.Full_Access_Only_Types
MD5: 6e7d4ee2e89b943d25319de9d8cebdcd

Likewise, we can define nonatomic and atomic full-access objects:

Listing 552: show_full_access_only_objects.ads

```

1 package Show_Full_Access_Only_Objects is
2
3   Nonatomic_Full_Access_Obj : Long_Float
4       with Volatile, Full_Access_Only;

```

(continues on next page)

(continued from previous page)

```

5   Atomic_Full_Access_Obj : Long_Float
6   with Atomic, Full_Access_Only;
7
8
9 end Show_Full_Access_Only_Objects;
```

Relevant topics

- 9.10 Shared Variables⁸⁴
- C.6 Shared Variable Control⁸⁵

Nonatomic full-access

As we already know, the value of a volatile object may be constantly changing, so the compiler generates code to read the value of the volatile object from memory for each access. (In other words, the value cannot be stored in a register for further processing.)

In the case of nonatomic full-access objects, the value of the object must not only be read from memory or updated to memory every time, but those operations must also be performed for the complete record object — not just parts of it.

Consider the following example:

Listing 553: registers.ads

```

1 with System;
2
3 package Registers is
4
5   type Boolean_Bit is new Boolean
6   with Size => 1;
7
8   type UInt1 is mod 2**1
9   with Size => 1;
10
11  type UInt2 is mod 2**2
12  with Size => 2;
13
14  type UInt14 is mod 2**14
15  with Size => 14;
16
17  type Window_Register is record
18    -- horizontal line count
19    Horizontal_Cnt : UInt14 := 16#0#;
20
21    -- unspecified
22    Reserved_14_15 : UInt2 := 16#0#;
23
24    -- vertical line count
25    Vertical_Cnt : UInt14 := 16#0#;
26
27    -- refresh signalling
28    Refresh_Needed : Boolean_Bit := False;
29
30    -- unspecified
31    Reserved_30 : UInt1 := 16#0#;
```

(continues on next page)

⁸⁴ <http://www.ada-auth.org/standards/22rm/html/RM-9-10.html>

⁸⁵ <http://www.ada-auth.org/standards/22rm/html/RM-C-6.html>

(continued from previous page)

```

32  end record
33  with Size      => 32,
34      Bit_Order => System.Low_Order_First,
35      Volatile,
36      Full_Access_Only;
37
38  for Window_Register use record
39      Horizontal_Cnt at 0 range 0 .. 13;
40      Reserved_14_15 at 0 range 14 .. 15;
41      Vertical_Cnt   at 0 range 16 .. 29;
42      Refresh_Needed at 0 range 30 .. 30;
43      Reserved_30    at 0 range 31 .. 31;
44  end record;
45
46  procedure Show (WR : Window_Register);
47
48  end Registers;

```

Listing 554: registers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Registers is
4
5      procedure Show (WR : Window_Register) is
6      begin
7          Put_Line ("WR = (Horizontal_Cnt => "
8                  & WR.Horizontal_Cnt'Image
9                  & ",");
10         Put_Line ("      Vertical_Cnt   => "
11                 & WR.Vertical_Cnt'Image
12                 & ",");
13         Put_Line ("      Refresh_Needed => "
14                 & WR.Refresh_Needed'Image
15                 & ")");
16     end Show;
17
18 end Registers;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Nonatomic_Full_Access_Register
 MD5: b825ec2dbbc54201203bf71e4e32fb57

In this example, we have a 32-bit register (of `Window_Register` type) that contains window information for a display:

position	0			
bits	#0 .. 13	#14 .. #15	#16 .. #29	#30 .. #31
component	Horizontal_Cnt	Reserved_14_15	Vertical_Cnt	Reserved_30_31

Let's use the `Window_Register` type from the `Registers` package in a test application:

Listing 555: show_register.adb

```

1  with Registers;   use Registers;
2
3  procedure Show_Register is
4      WR : Window_Register;
5  begin
6      -- Nonatomic full-access assignments
7      WR.Horizontal_Cnt := 800;
8      WR.Vertical_Cnt   := 600;
9      WR.Refresh_Needed := True;
10
11     Show (WR);
12 end Show_Register;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Nonatomic_Full_Access_Register
 MD5: 7ff302d6cb282a6276747e8e17f26dfd

Runtime output

```

WR = (Horizontal_Cnt => 800,
      Vertical_Cnt   => 600,
      Refresh_Needed => TRUE)

```

The example contains assignments such as `WR.Horizontal_Cnt := 800` and `WR.Vertical_Cnt := 600`. Because `Window_Register` is a full-access type, these assignments are performed for the complete 32-bit register, even though we're updating just a single component of the record object.

Note that if `Window_Register` wasn't a *full-access* object, an assignment such as `WR.Horizontal_Cnt := 800` could be performed with a 16-bit operation. In fact, this is what a compiler would most probably select for this assignment, because that is more efficient than manipulating the entire object. Therefore, using a *full-access* object prevents the compiler from generating operations that could lead to unexpected results.

Whenever possible, this *full-access* assignment is performed in a single machine operation. However, if it's not possible to generate a single machine operation on the target machine, the compiler may generate multiple operations for the update of the record components.

Note that we could combine these two assignments into a single one using an aggregate:

Listing 556: show_register.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Registers;   use Registers;
4
5  procedure Show_Register is
6      WR : Window_Register;
7  begin
8      -- Nonatomic full-access assignment
9      -- using an aggregate:
10     WR := (Horizontal_Cnt => 800,
11           Vertical_Cnt   => 600,
12           Refresh_Needed => True,
13           others          => <>);
14
15     Show (WR);
16 end Show_Register;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Nonatomic_Full_Access_Register
 MD5: 9caf39e4a01ee1ec62f0b24747640c01

Runtime output

```
WR = (Horizontal_Cnt => 800,
      Vertical_Cnt   => 600,
      Refresh_Needed => TRUE)
```

Again, this assignment is performed for the complete 32-bit register — ideally, using a single 32-bit machine operation — by reading the value from the memory.

Let's add another statement to the code example:

Listing 557: show_register.adb

```
1 with Registers; use Registers;
2
3 procedure Show_Register is
4   WR : Window_Register :=
5       (Horizontal_Cnt => 800,
6        Vertical_Cnt   => 600,
7        Refresh_Needed => True,
8        others         => <>);
9 begin
10  WR := (Horizontal_Cnt =>
11        WR.Horizontal_Cnt * 2,
12        Vertical_Cnt   =>
13        Wr.Vertical_Cnt * 2,
14        others         => <>);
15
16  Show (WR);
17 end Show_Register;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Nonatomic_Full_Access_Register
 MD5: cc4e218aef11af34e6d3262084a5c9ce

Runtime output

```
WR = (Horizontal_Cnt => 1600,
      Vertical_Cnt   => 1200,
      Refresh_Needed => FALSE)
```

In this example, we have an initialization using the same aggregate as in the previous code example. We also have an assignment, in which we read the value of WR and use it in the calculation.

Delta aggregates

If we want to just change two components, but leave the information of other components untouched, we can use a *delta aggregate* (page 557). (Note that we haven't discussed the topic of delta aggregates yet: we'll do that *later on in this course* (page 557). However, in simple terms, we can use them to modify specific components of a record without changing the remaining components of the record.)

For example, we might want to update just the vertical count and indicate that update via the Refresh_Needed flag, but keep the same horizontal count:

Listing 558: show_registers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Registers;   use Registers;
4
5  procedure Show_Registers is
6      WR : Window_Register :=
7          (Horizontal_Cnt => 800,
8           Vertical_Cnt   => 600,
9           others         => <>);
10
11  begin
12      -- Delta assignment
13      WR := (WR with delta
14             Vertical_Cnt   => 800,
15             Refresh_Needed => True);
16
17      Show (WR);
18  end Show_Registers;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
Control.Nonatomic_Full_Access_Register
MD5: 29df44d4fb13539cbd6070c37c217f8a

Runtime output

```

WR = (Horizontal_Cnt => 800,
      Vertical_Cnt   => 800,
      Refresh_Needed => TRUE)

```

A delta assignment using an aggregate such as (WR **with** delta ...) includes reading the value of the complete 32-bit WR object from memory, changing the components specified after **with** delta, and writing the complete 32-bit WR object back to memory. The reason is that we need to retrieve the information that is supposed to remain intact — the Horizontal_Cnt and the reserved components — in order to write them back as a *full-access* operation.

Atomic full-access

As we already know, *atomic objects* (page 414) only accept atomic reads and updates, which — as a whole — are indivisible, i.e. they must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes. (Again, if an object is atomic, this implies it is also volatile.)

In the case of atomic full-access objects, the complete object must be read and updated. Ideally, this operation corresponds to a single atomic operation on the target machine, but it can also translate to multiple atomic operations.

Let's adapt the previous example to illustrate this. First, we adapt the type in the package:

Listing 559: registers.ads

```

1  with System;
2
3  package Registers is
4
5      type Boolean_Bit is new Boolean
6          with Size => 1;
7

```

(continues on next page)

(continued from previous page)

```

8  type UInt1 is mod 2**1
9      with Size => 1;
10
11  type UInt2 is mod 2**2
12      with Size => 2;
13
14  type UInt14 is mod 2**14
15      with Size => 14;
16
17  type Window_Register is record
18      -- horizontal line count
19      Horizontal_Cnt : UInt14 := 16#0#;
20
21      -- unspecified
22      Reserved_14_15 : UInt2 := 16#0#;
23
24      -- vertical line count
25      Vertical_Cnt : UInt14 := 16#0#;
26
27      -- refresh signalling
28      Refresh_Needed : Boolean_Bit := False;
29
30      -- unspecified
31      Reserved_30 : UInt1 := 16#0#;
32  end record
33      with Size => 32,
34          Bit_Order => System.Low_Order_First,
35          Atomic,
36          Full_Access_Only;
37
38  for Window_Register use record
39      Horizontal_Cnt at 0 range 0 .. 13;
40      Reserved_14_15 at 0 range 14 .. 15;
41      Vertical_Cnt at 0 range 16 .. 29;
42      Refresh_Needed at 0 range 30 .. 30;
43      Reserved_30 at 0 range 31 .. 31;
44  end record;
45
46  procedure Show (WR : Window_Register);
47
48  end Registers;

```

Listing 560: registers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Registers is
4
5      procedure Show (WR : Window_Register) is
6      begin
7          Put_Line ("WR = (Horizontal_Cnt => "
8              & WR.Horizontal_Cnt'Image
9              & ",");
10         Put_Line ("      Vertical_Cnt => "
11             & WR.Vertical_Cnt'Image
12             & ",");
13         Put_Line ("      Refresh_Needed => "
14             & WR.Refresh_Needed'Image
15             & ")");
16     end Show;
17

```

(continues on next page)

(continued from previous page)

18 `end Registers;`**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Atomic_Full_Access_Register
 MD5: dc088d1b0df1af5086a1ae8b46bb6d4d

We then use the package in our test application:

Listing 561: show_register.adb

```

1  with Registers;   use Registers;
2
3  procedure Show_Register is
4      WR : Window_Register :=
5          (Horizontal_Cnt => 800,
6           Vertical_Cnt   => 600,
7           Refresh_Needed => True,
8           others         => <>);
9
10 begin
11     WR := (Horizontal_Cnt =>
12           WR.Horizontal_Cnt * 2,
13           Vertical_Cnt   =>
14           Wr.Vertical_Cnt  * 2,
15           others         => <>);
16
17     Show (WR);
18 end Show_Register;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Type_Representation.Shared_Variable_
 ↪Control.Atomic_Full_Access_Register
 MD5: cc4e218aef11af34e6d3262084a5c9ce

In this example, we first have an atomic initialization of WR using an aggregate. Then, we have an atomic assignment to the atomic full-access object WR. Because its type is an atomic full-access type, the operations are atomic operations that always access the full object from and to memory.

Comparison: full-access and non-full-access types

An interesting exercise for the reader is to compare the Assembly code generated for the code example above with a version of this code where the Window_Register is not a full-access type.

i Relevant topics

On a Linux platform, you can use *objdump* to retrieve the Assembly code and *diff* to see the difference between both versions of the type. For example:

```
objdump --target=elf64-x86-64 -d -S ./show_register > full_access.txt
```

```
# [...]
```

```
diff --width=80 -t -y full_access.txt no_full_access.txt
```

By doing this kind of comparisons, you might gain more insights on the impact of the `Full_Access_Only` aspect.

For further reading...

By running on a PC, we can compare the [Intel Assembly](#)⁸⁶ code for various versions of the code. Let's start with the version using a nonatomic full-access version of `Window_Register` vs. the nonatomic (non-full-access) version of `Window_Register`:

```
type Window_Register is record
  -- [...]
end record
with Size      => 32,
     Bit_Order => System.Low_Order_First,
     Volatile,
     Full_Access_Only;

type Window_Register is record
  -- [...]
end record
with Size      => 32,
     Bit_Order => System.Low_Order_First,
     Volatile;
```

These are the manually-adapted differences between both versions:

-- Volatile, Full_Access_Only		-- Volatile
<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x20,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc00,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax mov %eax,-0x14(%rbp) mov -0x14(%rbp),%eax mov %eax,-0x8(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x8(%rbp),%eax mov %eax,%ecx </pre>		<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x10,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc00,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x4(%rbp),%eax </pre>
	<	
	<	
	<	
	<	

and \$0x3fff,%cx		and \$0x3fff,%ax
WR := (Horizontal_Cnt =>	>	add %eax,%eax
mov -0xc(%rbp),%eax	<	
mov %eax,%edx	<	
WR.Horizontal_Cnt * 2,	<	
lea (%rcx,%rcx,1),%eax	<	
and \$0x3fff,%ax		and \$0x3fff,%ax
WR := (Horizontal_Cnt =>		WR := (Horizontal_Cnt =>
movzwl %ax,%eax		movzwl %ax,%eax
and \$0x3fff,%eax		and \$0x3fff,%eax
and \$0xffffc000,%edx	<	
or %edx,%eax	<	
mov %eax,%edx		mov %eax,%edx
mov %edx,%eax		mov -0x8(%rbp),%eax
mov %eax,-0xc(%rbp)		and \$0xffffc000,%eax
mov -0xc(%rbp),%eax		or %edx,%eax
	>	mov %eax,-0x8(%rbp)
	>	mov -0x8(%rbp),%eax
		and \$0x3f,%ah
and \$0x3f,%ah		mov %eax,-0x8(%rbp)
mov %eax,-0xc(%rbp)		Vertical_Cnt =>
Vertical_Cnt =>		Wr.Vertical_Cnt * 2,
Wr.Vertical_Cnt * 2,		mov -0x4(%rbp),%eax
mov -0x8(%rbp),%eax		shr \$0x10,%eax
shr \$0x10,%eax		and \$0x3fff,%ax
mov %eax,%ecx		add %eax,%eax
and \$0x3fff,%cx		
WR := (Horizontal_Cnt =>	<	
mov -0xc(%rbp),%eax	<	
mov %eax,%edx	<	
Wr.Vertical_Cnt * 2,	<	
lea (%rcx,%rcx,1),%eax	<	
and \$0x3fff,%ax		and \$0x3fff,%ax
WR := (Horizontal_Cnt =>		WR := (Horizontal_Cnt =>
movzwl %ax,%eax		movzwl %ax,%eax
and \$0x3fff,%eax		and \$0x3fff,%eax
shl \$0x10,%eax		shl \$0x10,%eax
and \$0xc000ffff,%edx	<	
or %edx,%eax	<	
mov %eax,%edx		mov %eax,%edx
mov %edx,%eax		mov -0x8(%rbp),%eax
mov %eax,-0xc(%rbp)		and \$0xc000ffff,%eax
mov -0xc(%rbp),%eax		or %edx,%eax
	>	mov %eax,-0x8(%rbp)
	>	mov -0x8(%rbp),%eax
		and \$0xbfffffff,%eax
and \$0xbfffffff,%eax		mov %eax,-0x8(%rbp)
mov %eax,-0xc(%rbp)		mov -0x8(%rbp),%eax
mov -0xc(%rbp),%eax		mov -0x8(%rbp),%eax
and \$0x7fffffff,%eax	<	and \$0x7fffffff,%eax
mov %eax,-0xc(%rbp)	<	
mov -0xc(%rbp),%eax		mov %eax,-0x8(%rbp)
mov %eax,-0x8(%rbp)	>	mov -0x8(%rbp),%eax
	>	mov %eax,-0x4(%rbp)
others => <>;		others => <>;

As we can see, although parts of the Assembly code are the same or look very similar, there are some differences between both versions. These differences are mostly related to the fact that we have to operate on the full object when reading it from memory.

Likewise, we can compare the Assembly code for the atomic full-access version of Window_Register vs. the atomic (non-full-access) version of Window_Register:

```

type Window_Register is record
  -- [...]
end record
with Size      => 32,
     Bit_Order => System.Low_Order_First,
     Atomic,
     Full_Access_Only;

type Window_Register is record
  -- [...]
end record
with Size      => 32,
     Bit_Order => System.Low_Order_First,
     Atomic;

```

These are the manually-adapted differences between both versions:

-- Atomic, Full_Access_Only		-- Atomic
<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x20,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc00,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) WR : Window_Register := mov -0x4(%rbp),%eax mov %eax,-0x14(%rbp) mov -0x14(%rbp),%eax mov %eax,-0x8(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x8(%rbp),%eax mov %eax,%ecx and \$0x3fff,%cx WR := (Horizontal_Cnt => mov -0xc(%rbp),%eax mov %eax,%edx WR.Horizontal_Cnt * 2, lea (%rcx,%rcx,1),%eax and \$0x3fff,%ax </pre>		
		<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x10,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc00,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) WR : Window_Register := mov -0x4(%rbp),%eax mov %eax,-0x8(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x8(%rbp),%eax and \$0x3fff,%ax add %eax,%eax and \$0x3fff,%ax </pre>

WR := (Horizontal_Cnt =>		WR := (Horizontal_Cnt =>
movzwl %ax,%eax		movzwl %ax,%eax
and \$0x3fff,%eax		and \$0x3fff,%eax
	>	mov %eax,%edx
	>	mov -0xc(%rbp),%eax
and \$0xffffc000,%edx		and \$0xffffc000,%eax
or %edx,%eax		or %edx,%eax
mov %eax,%edx	<	
mov %edx,%eax	<	mov %eax,-0xc(%rbp)
mov %eax,-0xc(%rbp)		mov -0xc(%rbp),%eax
mov -0xc(%rbp),%eax		and \$0x3f,%ah
and \$0x3f,%ah		mov %eax,-0xc(%rbp)
mov %eax,-0xc(%rbp)		Vertical_Cnt =>
Vertical_Cnt =>		Wr.Vertical_Cnt * 2,
Wr.Vertical_Cnt * 2,		mov %eax,-0x8(%rbp),%eax
mov -0x8(%rbp),%eax		shr \$0x10,%eax
shr \$0x10,%eax		
mov %eax,%ecx	<	shr \$0x10,%eax
and \$0x3fff,%cx		
	>	and \$0x3fff,%ax
WR := (Horizontal_Cnt =>	<	add %eax,%eax
mov -0xc(%rbp),%eax	<	
mov %eax,%edx	<	
Wr.Vertical_Cnt * 2,	<	
lea (%rcx,%rcx,1),%eax	<	and \$0x3fff,%ax
and \$0x3fff,%ax		WR := (Horizontal_Cnt =>
WR := (Horizontal_Cnt =>		movzwl %ax,%eax
movzwl %ax,%eax		and \$0x3fff,%eax
and \$0x3fff,%eax		shl \$0x10,%eax
shl \$0x10,%eax		mov %eax,%edx
	>	mov -0xc(%rbp),%eax
	>	and \$0xc000ffff,%eax
and \$0xc000ffff,%edx		or %edx,%eax
or %edx,%eax		
mov %eax,%edx	<	
mov %edx,%eax	<	mov %eax,-0xc(%rbp)
mov %eax,-0xc(%rbp)		mov -0xc(%rbp),%eax
mov -0xc(%rbp),%eax		and \$0xbfffffff,%eax
and \$0xbfffffff,%eax		mov %eax,-0xc(%rbp)
mov %eax,-0xc(%rbp)		mov -0xc(%rbp),%eax
mov -0xc(%rbp),%eax		and \$0x7fffffff,%eax
and \$0x7fffffff,%eax		mov %eax,-0xc(%rbp)
mov %eax,-0xc(%rbp)		mov -0xc(%rbp),%eax
mov -0xc(%rbp),%eax		mov %eax,-0xc(%rbp)
xchg %eax,-0x8(%rbp)		xchg %eax,-0x8(%rbp)
others => <>;		others => <>;

Again, there are some differences between both versions, even though some parts of the Assembly code are the same or look very similar.

Finally, we might want to compare the nonatomic full-access version vs. the atomic full-access version of the Window Register type:

```

type Window_Register is record
    -- [...]
end record
    with Size           => 32,
         Bit_Order      => System.Low_Order_First,
         Volatile,
         Full_Access_Only;

type Window_Register is record
    -- [...]
end record
    with Size           => 32,
         Bit_Order      => System.Low_Order_First,
         Atomic,
         Full Access Only;

```

These are the differences between both versions:

-- Volatile, Full_Access_Only		-- Atomic, Full_Access_Only
<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x20,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc000,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) WR : Window_Register := mov -0x4(%rbp),%eax mov %eax,-0x14(%rbp) mov -0x14(%rbp),%eax mov %eax,-0x8(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x8(%rbp),%eax mov %eax,%ecx and \$0x3fff,%cx WR := (Horizontal_Cnt => mov -0xc(%rbp),%eax mov %eax,%edx WR.Horizontal_Cnt * 2, lea (%rcx,%rcx,1),%eax and \$0x3fff,%ax WR := (Horizontal_Cnt => movzwl %ax,%eax and \$0x3fff,%eax and \$0xfffffc000,%edx or %edx,%eax mov %eax,%edx mov %edx,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0x3f,%ah mov %eax,-0xc(%rbp) Vertical_Cnt => Wr.Vertical_Cnt * 2, mov -0x8(%rbp),%eax shr \$0x10,%eax mov %eax,%ecx </pre>		
<pre> procedure Show_Register is push %rbp mov %rsp,%rbp sub \$0x20,%rsp WR : Window_Register := (Horizontal_Cnt => 800, mov -0x4(%rbp),%eax and \$0xfffffc000,%eax or \$0x320,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x3f,%ah mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0xc000ffff,%eax or \$0x2580000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax or \$0x40000000,%eax mov %eax,-0x4(%rbp) mov -0x4(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0x4(%rbp) WR : Window_Register := mov -0x4(%rbp),%eax mov %eax,-0x14(%rbp) mov -0x14(%rbp),%eax mov %eax,-0x8(%rbp) Vertical_Cnt => 600, Refresh_Needed => True, others => <>); begin WR := (Horizontal_Cnt => WR.Horizontal_Cnt * 2, mov -0x8(%rbp),%eax mov %eax,%ecx and \$0x3fff,%cx WR := (Horizontal_Cnt => mov -0xc(%rbp),%eax mov %eax,%edx WR.Horizontal_Cnt * 2, lea (%rcx,%rcx,1),%eax and \$0x3fff,%ax WR := (Horizontal_Cnt => movzwl %ax,%eax and \$0x3fff,%eax and \$0xfffffc000,%edx or %edx,%eax mov %eax,%edx mov %edx,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0x3f,%ah mov %eax,-0xc(%rbp) Vertical_Cnt => Wr.Vertical_Cnt * 2, mov -0x8(%rbp),%eax shr \$0x10,%eax mov %eax,%ecx </pre>		

<pre> and \$0x3fff,%cx WR := (Horizontal_Cnt => mov -0xc(%rbp),%eax mov %eax,%edx Wr.Vertical_Cnt * 2, lea (%rcx,%rcx,1),%eax and \$0x3fff,%ax WR := (Horizontal_Cnt => movzwl %ax,%eax and \$0x3fff,%eax shl \$0x10,%eax and \$0xc000ffff,%edx or %edx,%eax mov %eax,%edx mov %edx,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0xbfffffff,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax mov %eax,-0x8(%rbp) others => <>; </pre>	<pre> and \$0x3fff,%cx WR := (Horizontal_Cnt => mov -0xc(%rbp),%eax mov %eax,%edx Wr.Vertical_Cnt * 2, lea (%rcx,%rcx,1),%eax and \$0x3fff,%ax WR := (Horizontal_Cnt => movzwl %ax,%eax and \$0x3fff,%eax shl \$0x10,%eax and \$0xc000ffff,%edx or %edx,%eax mov %eax,%edx mov %edx,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0xbfffffff,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax and \$0x7fffffff,%eax mov %eax,-0xc(%rbp) mov -0xc(%rbp),%eax mov %eax,-0x8(%rbp) xchg %eax,-0x8(%rbp) others => <>; </pre>

As we can see, the code is basically the same — except for the last Assembly instruction, which is a *mov* instruction in the volatile version and an *xchg* instruction in the atomic version — which is an atomic instruction on this platform.

25.3.5 Atomic operations

Note

This feature was introduced in Ada 2022.

Ada offers four packages to handle atomic operations. Those packages are child packages of the `System.Atomic_Operations` package. We will discuss each of those package individually in this section.

Relevant topics

- [C.6.1 The Package System.Atomic_Operations](#)⁸⁷

Atomic Exchange

The generic `System.Atomic_Operations.Exchange` package provides operations to compare and exchange objects atomically.

⁸⁶ https://en.wikipedia.org/wiki/X86_instruction_listings

⁸⁷ <http://www.ada-auth.org/standards/22rm/html/RM-C-6-1.html>

Atomic_Exchange function

One of those operations is the Atomic_Exchange function, which performs the following operations atomically:

```
function Atomic_Exchange
(Item : aliased in out Atomic_Type;
 Value : Atomic_Type)
return Atomic_Type
is
  Old_Item : Atomic_Type := Item;
begin
  Item := Value;
  return Old_Item;
end Atomic_Exchange;
```

As mentioned in the [Ada Reference Manual](#)⁸⁸, we can use this function to implement a [spinlock](#)⁸⁹. For example:

Listing 562: spinlocks.ads

```
1 with System.Atomic_Operations.Exchange;
2
3 package Spinlocks is
4
5   type Lock is new Boolean with Atomic;
6
7   package Lock_Exchange is new
8     System.Atomic_Operations.Exchange (Lock);
9
10 end Spinlocks;
```

Listing 563: show_locks.adb

```
1 with Spinlocks;
2 use Spinlocks;
3 use Spinlocks.Lock_Exchange;
4
5 procedure Show_Locks is
6   L : aliased Lock := False;
7 begin
8   -- Get the lock
9   while Atomic_Exchange (Item => L,
10                        Value => True) loop
11     null;
12   end loop;
13
14   -- At this point, we got the lock.
15   -- Do some stuff here...
16
17   -- Release the lock.
18   L := False;
19 end Show_Locks;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪ Exchange
 MD5: 36699b917485f14c4e8a905a6c48027b

⁸⁸ <http://www.ada-auth.org/standards/22rm/html/RM-C-6-2.html>

⁸⁹ <https://en.wikipedia.org/wiki/Spinlock>

In this example, we call the `Atomic_Exchange` function for the `L` lock until we get it. Then, we can use the resource that we protected via the lock. After we finish our work, we can release the lock by setting `L` to **False**.

Note that `System.Atomic_Operations.Exchange` is a generic package, so we have to instantiate it for a specific atomic type — in this case, the atomic Boolean Lock type.

We can use multiple tasks to illustrate a situation where using a lock is important to ensure that no **race conditions**⁹⁰ occur:

Listing 564: spinlocks.ads

```
1 with System.Atomic_Operations.Exchange;
2
3 package Spinlocks is
4
5     type Lock is new Boolean with Atomic;
6
7     package Lock_Exchange is new
8         System.Atomic_Operations.Exchange (Lock);
9
10 end Spinlocks;
```

Listing 565: show_locks.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Spinlocks;
4 use Spinlocks;
5 use Spinlocks.Lock_Exchange;
6
7 procedure Show_Locks is
8     L      : aliased Lock := False;
9     Task_Count : Integer := 0;
10
11     task type A_Task;
12
13     task body A_Task is
14         Task_Number : Integer;
15     begin
16         -- Get the lock
17         while Atomic_Exchange (Item => L,
18                               Value => True) loop
19             null;
20         end loop;
21
22         -- At this point, we got the lock.
23         Task_Count := Task_Count + 1;
24         Task_Number := Task_Count;
25
26         -- Release the lock.
27         L := False;
28
29         Put_Line ("Task_Number: "
30                 & Task_Number'Image);
31
32     end A_Task;
33
34     A, B, C, D, E, F : A_Task;
35 begin
36     null;
```

(continues on next page)

⁹⁰ https://en.wikipedia.org/wiki/Race_condition

(continued from previous page)

37 `end Show_Locks;`**Code block metadata**

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪Exchange
 MD5: af7aad741c20be1e8433b04def90dcdb

Runtime output

```
Task_Number: 1
Task_Number: 2
Task_Number: 5
Task_Number: 6
Task_Number: 3
Task_Number: 4
```

In this example, we create multiple tasks (A, B, C, D, E, F) and initialize the **Task_Number** of each task based on the value of the **Task_Count** variable. To avoid multiple tasks accessing the **Task_Count** variable at the same time, we use the L lock, which we get before updating the **Task_Count**.

Atomic_Compare_And_Exchange function

Another function from the `System.Atomic_Operations.Exchange` package is `Atomic_Compare_And_Exchange`, which performs the following operations atomically:

```
function Atomic_Compare_And_Exchange
(Item      : aliased in out Atomic_Type;
 Prior     : aliased in out Atomic_Type;
 Desired   : Atomic_Type)
return Boolean is
begin
  if Item = Prior then
    Item := Value;
    -- The item is only updated if its
    -- value and the prior value match

    return True;
  else
    Prior := Item;
    return False;
  end if;
end Atomic_Exchange;
```

This function can be used for **lazy initialization**⁹¹ of variables. For example, consider an application with multiple tasks that make use of a certain value that isn't initialized at its declaration, but at a later point in time by an arbitrary task. We can use `Atomic_Compare_And_Exchange` to ensure that we only update that value if it wasn't already initialized.

Let's start with the package specification:

Listing 566: `lazy_initialization.ads`

```
1 with System.Atomic_Operations.Exchange;
2 with Ada.Numerics.Discrete_Random;
3
```

(continues on next page)

⁹¹ https://en.wikipedia.org/wiki/Lazy_initialization

(continued from previous page)

```

4 package Lazy_Initialization is
5
6     subtype Lazy_Value_Total_Range is
7         Integer range 99 .. 1000;
8
9     Lazy_Value_Default_Value : constant
10        := Lazy_Value_Total_Range'First;
11
12     subtype Lazy_Value_Range is Integer
13         range Lazy_Value_Default_Value + 1 ..
14             Lazy_Value_Total_Range'Last;
15
16     type Lazy_Value is new Lazy_Value_Total_Range
17         with Atomic,
18             Default_Value =>
19                 Lazy_Value_Default_Value;
20
21     package Value_Exchange is new
22         System.Atomic_Operations.Exchange
23         (Lazy_Value);
24
25     package Lazy_Value_Random is new
26         Ada.Numerics.Discrete_Random
27         (Lazy_Value_Range);
28
29 end Lazy_Initialization;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪ Compare_And_Exchange
 MD5: 09d49998aa7e3d5c0cfb4b74af8e542b

In this package, we declare the `Lazy_Value` type with a default value (specified by the `Lazy_Value_Default_Value` constant). Note that we have two ranges here: `Lazy_Value_Total_Range` and `Lazy_Value_Range`. We use the `Lazy_Value_Total_Range` in the declaration of the `Lazy_Value` type: it indicates the *total range* of the type. We use the `Lazy_Value_Range` as a constraint for the total range. This range doesn't contain the default value (`Lazy_Value_Default_Value`), and we use it to indicate the valid values of the type. (We discuss the application of `Lazy_Value_Range` later on.)

Also, in addition to instantiating the `System.Atomic_Operations.Exchange` package, we instantiate the `Ada.Numerics.Discrete_Random` package, which we'll use to generate random numbers in the expected range (`Lazy_Value_Range`) for the `Lazy_Value` type. (We discussed the `Ada.Numerics.Discrete_Random` package in the [Introduction to Ada](#) (page 262) course.)

Let's use this package in the `Show_Lazy_Initialization` procedure:

Listing 567: show_lazy_initialization.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics.Discrete_Random;
3
4 with Lazy_Initialization;
5 use Lazy_Initialization;
6 use Lazy_Initialization.Value_Exchange;
7
8 procedure Show_Lazy_Initialization is
9     subtype A_Task_Number is Natural;
10

```

(continues on next page)

(continued from previous page)

```

11 Value          : aliased Lazy_Value;
12 Value_Modified_By : A_Task_Number := 0;
13
14 task type A_Task is
15     entry Start (This : A_Task_Number);
16     entry Stop;
17 end A_Task;
18
19 task body A_Task is
20     Task_Number : A_Task_Number;
21 begin
22     accept Start (This : A_Task_Number) do
23         Task_Number := This;
24     end Start;
25
26     Sleep_Some_Time : declare
27         subtype Sleep_Range is
28             Integer range 1 .. 3;
29
30         package Random_Sleep is new
31             Ada.Numerics.Discrete_Random
32                 (Sleep_Range);
33         use Random_Sleep;
34
35         G : Generator;
36     begin
37         Reset (G);
38         delay Duration (Random (G));
39     end Sleep_Some_Time;
40
41     Generate_Value : declare
42         use Lazy_Value_Random;
43
44         G          : Generator;
45         Initial_Value : Lazy_Value_Range;
46         Prior       : aliased Lazy_Value;
47     begin
48         Reset (G);
49         Initial_Value := Random (G);
50
51         if Atomic_Compare_And_Exchange
52             (Item    => Value,
53              Prior   => Prior,
54              Desired => Lazy_Value (Initial_Value))
55         then
56             Value_Modified_By := Task_Number;
57         end if;
58
59     end Generate_Value;
60
61     accept Stop do
62         Put_Line ("Current task number: "
63                 & Task_Number'Image);
64         Put_Line ("Value: "
65                 & Value'Image);
66         Put_Line ("Modified by task number: "
67                 & Value_Modified_By'Image);
68         Put_Line ("-----");
69     end Stop;
70 end A_Task;
71

```

(continues on next page)

(continued from previous page)

```

72   Some_Tasks : array (1 .. 5) of A_Task;
73 begin
74   for I in Some_Tasks'Range loop
75     Some_Tasks (I).Start (I);
76   end loop;
77   for I in Some_Tasks'Range loop
78     Some_Tasks (I).Stop;
79   end loop;
80 end Show_Lazy_Initialization;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↳ Compare_And_Exchange
 MD5: 9cc898edd767f8bbcfce2c81e7ca0e442

Runtime output

```

Current task number: 1
Value: 636
Modified by task number: 5
-----
Current task number: 2
Value: 636
Modified by task number: 5
-----
Current task number: 3
Value: 636
Modified by task number: 5
-----
Current task number: 4
Value: 636
Modified by task number: 5
-----
Current task number: 5
Value: 636
Modified by task number: 5
-----

```

In the `Show_Lazy_Initialization` procedure, the most important variable is `Value`, which is the variable we have to protect via a lock. In addition, we have the auxiliary `Value_Modified_By` variable, which indicates the number of the task that initialized the `Value` variable.

In this procedure, we also see two main *block statements* (page 758):

- the block statement with the `Sleep_Some_Time` identifier, where we make the task *sleep* for a random amount of time (in the `Sleep_Range` range); and
- the block statement with the `Generate_Value` identified, where we generate a new value randomly and attempt to update the `Value` variable (of `Lazy_Value` type).

Let's discuss some details about the `Generate_Value` block statement. We start by declaring some variables. Here, it's important to highlight that the `Prior` variable is initialized with the default value (`Lazy_Value_Default_Value`). We then call the `Atomic_Compare_And_Exchange` function, and pass `Value` and `Prior` as actual parameters. We can have two possible outcomes:

1. If `Value` hasn't been modified by a task yet, it will contain the default value — which means that the values of the `Prior` and `Value` variables match. In this case, the call to `Atomic_Compare_And_Exchange` will update the `Value` variable and return **True**. (Note that we also update the `Value_Modified_By` variable when

Atomic_Compare_And_Exchange returns **True**.)

2. If Value has already been modified by a task, its value doesn't match the (default) value of Prior anymore, so the call to Atomic_Compare_And_Exchange doesn't modify the Value variable.

As mentioned before, we use a stricter range for the random number generator: the Lazy_Value_Range. Because this range doesn't contain the default value (Lazy_Value_Default_Value), we will never generate a random value that matches the default value.

i Relevant topics

- C.6.2 The Package System.Atomic_Operations.Exchange⁹²

Atomic Test and Set

The System.Atomic_Operations.Test_And_Set package provides atomic operations to set and clear atomic flags. To declare flags, we use the Test_And_Set_Flag type. The following operations are available:

1. the Atomic_Test_And_Set function, which we call to verify whether the flag can be set and, if positive, set it accordingly.
 - The function returns **True** if the flag has been set, and **False** otherwise.
2. the Atomic_Clear procedure, which we call to clear the flag.

We can use these functions to implement an application similar to the *spinlocks* (page 431) that we've seen before:

Listing 568: show_test_and_set.adb

```

1 with System.Atomic_Operations.Test_And_Set;
2 use System.Atomic_Operations.Test_And_Set;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Test_And_Set is
7   Lock      : aliased Test_And_Set_Flag;
8   Task_Count : Integer := 0;
9
10  task type A_Task;
11
12  task body A_Task is
13    Task_Number : Integer;
14  begin
15    -- Get the lock
16    while Atomic_Test_And_Set (Lock) loop
17      null;
18    end loop;
19
20    -- At this point, we got the lock.
21    Task_Count := Task_Count + 1;
22    Task_Number := Task_Count;
23
24    -- Release the lock.
25    Atomic_Clear (Lock);
26
27    Put_Line ("Task_Number: "
```

(continues on next page)

⁹² <http://www.ada-auth.org/standards/22rm/html/RM-C-6-2.html>

(continued from previous page)

```

28         & Task_Number'Image);
29
30     end A_Task;
31
32     A, B, C, D, E, F : A_Task;
33 begin
34     null;
35 end Show_Test_And_Set;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪ Test_And_Set
 MD5: 45814e2e157d3fd45f876c89914a5cc5

Runtime output

```

Task_Number: 2
Task_Number: 3
Task_Number: 4
Task_Number: 5
Task_Number: 6
Task_Number: 1

```

Here, we call `Atomic_Test_And_Set` in a loop until it returns **True**. Then, we update the **Task_Count** and **Task_Number**. When we're finished, we call the `Atomic_Clear` procedure to release the lock.

Relevant topics

- C.6.3 The Package `System.Atomic_Operations.Test_and_Set`⁹³

Atomic Operations using Integer Arithmetic

The generic `System.Atomic_Operations.Integer_Arithmetic` package is used to perform atomic operations on atomic integer types. It provides the following operations: the procedures `Atomic_Add` and `Atomic_Subtract`, and the functions `Atomic_Fetch_And_Add` and `Atomic_Fetch_And_Subtract`. The procedures and the corresponding `Atomic_Fetch` functions do basically the same thing, with the difference that `Atomic_Fetch` functions return the previous (older) value of the input item.

The `Atomic_Add` procedure performs the following operations atomically:

```

procedure Atomic_Add
  (Item : aliased in out Atomic_Type;
   Value : Atomic_Type) is
begin
  Item := Item + Value;
end Atomic_Add;

```

The corresponding `Atomic_Fetch_And_Add` function performs the following operations atomically:

```

function Atomic_Fetch_And_Add
  (Item : aliased in out Atomic_Type;
   Value : Atomic_Type)

```

(continues on next page)

⁹³ <http://www.ada-auth.org/standards/22rm/html/RM-C-6-3.html>

(continued from previous page)

```

    return Atomic_Type
is
    Old_Item : Atomic_Type := Item;
begin
    Item := Item + Value;
    return Old_Item;
end Atomic_Fetch_And_Add;

```

The Atomic_Subtract procedure performs the following operations atomically:

```

procedure Atomic_Subtract
(Item : aliased in out Atomic_Type;
 Value : Atomic_Type) is
begin
    Item := Item - Value;
end Atomic_Subtract;

```

The corresponding Atomic_Fetch_And_Subtract function performs the following operations atomically:

```

function Atomic_Fetch_And_Subtract
(Item : aliased in out Atomic_Type;
 Value : Atomic_Type)
return Atomic_Type
is
    Old_Item : Atomic_Type := Item;
begin
    Item := Item - Value;
    return Old_Item;
end Atomic_Fetch_And_Subtract;

```

Let's reuse a [previous code example](#) (page 432) that sets a unique number for each task. In this case, instead of using locks, we use the atomic operations from the System.Atomic_Operations.Integer_Arithmetic package:

Listing 569: atomic_integers.ads

```

1 with System.Atomic_Operations.Integer_Arithmetic;
2
3 package Atomic_Integers is
4
5     type Atomic_Integer is new Integer
6     with Atomic;
7
8     package Atomic_Integer_Arithmetic is new
9     System.Atomic_Operations.Integer_Arithmetic
10     (Atomic_Integer);
11
12 end Atomic_Integers;

```

Listing 570: show_atomic_integers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Atomic_Integers;
4 use Atomic_Integers;
5 use Atomic_Integers.Atomic_Integer_Arithmetic;
6
7 procedure Show_Atomic_Integers is
8     Task_Count : aliased Atomic_Integer := 0;

```

(continues on next page)

(continued from previous page)

```

9
10 task type A_Task;
11
12 task body A_Task is
13     Task_Number : Atomic_Integer;
14 begin
15     Task_Number :=
16         Atomic_Fetch_And_Add (Task_Count, 1);
17
18     Put_Line ("Task_Number: "
19             & Task_Number'Image);
20
21 end A_Task;
22
23 A, B, C, D, E, F : A_Task;
24 begin
25     null;
26 end Show_Atomic_Integers;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
Integer_Arithmetic
MD5: 835093f90b9efe37b93ca84fe1ce3444

Runtime output

```

Task_Number: 3
Task_Number: 4
Task_Number: 0
Task_Number: 5
Task_Number: 1
Task_Number: 2

```

In this example, we call the `Atomic_Fetch_And_Add` function to update the `Task_Count` variable and, at the same time, initialize the `Task_Number` variable of the current task.

Relevant topics

- C.6.4 The Package `System.Atomic_Operations.Integer_Arithmetic`⁹⁴

Atomic Operations using Modular Arithmetic

The generic `System.Atomic_Operations.Modular_Arithmetic` package is very similar to the `System.Atomic_Operations.Integer_Arithmetic` package. In fact, it provides the same operations: the procedures `Atomic_Add` and `Atomic_Subtract`, and the functions `Atomic_Fetch_And_Add` and `Atomic_Fetch_And_Subtract`. The only difference is that it is used for modular types instead of integer types.

Let's reuse the *previous code example* (page 439), but replace the atomic integer type by an atomic modular type:

Listing 571: `atomic_modulars.ads`

```

1 with System.Atomic_Operations.Modular_Arithmetic;
2
3 package Atomic_Modulars is

```

(continues on next page)

⁹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-C-6-4.html>

(continued from previous page)

```

4  type Atomic_Modular is mod 100
5      with Atomic;
6
7
8  package Atomic_Modular_Arithmetic is new
9      System.Atomic_Operations.Modular_Arithmetic
10         (Atomic_Modular);
11
12 end Atomic_Modulars;

```

Listing 572: show_atomic_modulars.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Atomic_Modulars;
4  use Atomic_Modulars;
5  use Atomic_Modulars.Atomic_Modular_Arithmetic;
6
7  procedure Show_Atomic_Modulars is
8      Task_Count : aliased Atomic_Modular := 0;
9
10     task type A_Task;
11
12     task body A_Task is
13         Task_Number : Atomic_Modular;
14     begin
15         Task_Number :=
16             Atomic_Fetch_And_Add (Task_Count, 1);
17
18         Put_Line ("Task_Number: "
19                 & Task_Number'Image);
20
21     end A_Task;
22
23     A, B, C, D, E, F : A_Task;
24 begin
25     null;
26 end Show_Atomic_Modulars;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Shared_Variable_Control.Atomic_Operations.
 ↪Modular_Arithmetic
 MD5: 3a5a85febacd13f5e053cf00b19746ff

Runtime output

```

Task_Number: 0
Task_Number: 2
Task_Number: 3
Task_Number: 1
Task_Number: 4
Task_Number: 5

```

As we did in the previous example, we again call the `Atomic_Fetch_And_Add` function to update the `Task_Count` variable and, at the same time, initialize the `Task_Number` variable of the current task. The only difference is that we use a modular type (`Atomic_Modular`).

** Relevant topics**

- C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic⁹⁵

25.4 Records

25.4.1 Default Initialization

As mentioned in the *Introduction to Ada* (page 65) course, record components can have default initial values. Also, we've seen that other kinds of types can have *default values* (page 339).

In the Ada Reference Manual, we refer to these default initial values as "default expressions of record components." The term *default expression* indicates that we can use any kind of expression for the default initialization of record components — which includes subprogram calls for example:

Listing 573: show_default_initialization.ads

```

1 package Show_Default_Initialization is
2
3     function Init return Integer is
4         (42);
5
6     type Rec is record
7         A : Integer := Init;
8     end record;
9
10 end Show_Default_Initialization;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Simple_Example
 MD5: 6d06be7f087513b669ba5481d6ee5004

In this example, the A component is initialized by default by a call to the Init procedure.

In the Ada Reference Manual

- 3.8 Record Types⁹⁶

Dependencies

Default expressions cannot depend on other components. For example, if we have two components A and B, we cannot initialize B based on the value that A has:

Listing 574: show_default_initialization_dependency.ads

```

1 package Show_Default_Initialization_Dependency is
2
3     function Init return Integer is
4         (42);
5
6     type Rec is record
7         A : Integer := Init;
```

(continues on next page)

⁹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-C-6-5.html>

⁹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

(continued from previous page)

```

8      B : Integer := Rec.A;  -- Illegal!
9      end record;
10
11 end Show_Default_Initialization_Dependency;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.No_
 ↳Dependency
 MD5: ca23cbd7e4a54d0b9c6974aed0ee77c8

Build output

```

show_default_initialization_dependency.ads:8:25: error: component "Rec.A" cannot
↳be used before end of record declaration
gprbuild: *** compilation phase failed

```

In this example, we cannot initialize the B component based on the value of the A component. (In fact, the syntax `Rec.A` as a way to refer to the A component is only allowed in predicates, not in the record component declaration.)

Initialization Order

The default initialization of record components is performed in arbitrary order. In fact, the order is decided by the compiler, so we don't have control over it.

Let's see an example:

Listing 575: simple_recs.ads

```

1  package Simple_Recs is
2
3      function Init (S : String;
4                    I : Integer)
5                      return Integer;
6
7      type Rec is record
8          A : Integer := Init ("A", 1);
9          B : Integer := Init ("B", 2);
10     end record;
11
12 end Simple_Recs;

```

Listing 576: simple_recs.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Recs is
4
5      function Init (S : String;
6                    I : Integer)
7                      return Integer is
8
9      begin
10         Put_Line (S & ": " & I'Image);
11         return I;
12     end Init;
13 end Simple_Recs;

```

Listing 577: show_initialization_order.adb

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Show_Initialization_Order is
4   R : Rec;
5 begin
6   null;
7 end Show_Initialization_Order;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.
Initialization_Order
MD5: e3ab92ea9b2a99815cea8c2ea11cbbfb

Runtime output

```
A: 1
B: 2
```

When running this code example, you might see this:

```
A: 1
B: 2
```

However, the compiler is allowed to rearrange the operations, so this output is possible as well:

```
B: 2
A: 1
```

Therefore, we must write the default expression of each individual record components in such a way that the resulting initialization value is always correct, independently of the order that those expressions are evaluated.

Evaluation

According to the Annotated Ada Reference Manual, the "default expression of a record component is only evaluated upon the creation of a default-initialized object of the record type." This means that the default expression is by itself not evaluated when we declare the record type, but when we create an object of this type. It follows from this rule that the default is only evaluated when necessary, i.e., when an explicit initial value is not specified in the object declaration.

Let's see an example:

Listing 578: show_initialization_order.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Simple_Recs; use Simple_Recs;
3
4 procedure Show_Initialization_Order is
5 begin
6   Put_Line ("Some processing first...");
7   Put_Line
8     ("Now, let's declare an object "
9      & "of the record type Rec...");
10
11   declare
```

(continues on next page)

(continued from previous page)

```

12     R : Rec;
13     begin
14         Put_Line
15             ("An object of Rec type has "
16              & "just been created.");
17     end;
18
19 end Show_Initialization_Order;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.
 ↪ Initialization_Order
 MD5: 126e3edfe4cb8033f40b939ff9922958

Runtime output

Some processing first...
 Now, let's declare an object of the record type Rec...
 A: 1
 B: 2
 An object of Rec type has just been created.

Here, we only see the information displayed by the Init function — which is called to initialize the A and B components of the R record — during the object creation. In other words, the default expressions Init ("A", 1) and Init ("B", 2) are *not* evaluated when we declare the R type, but when we create an object of this type.

i In the Ada Reference Manual

- 3.8 Record Types⁹⁷

Defaults and object declaration**i Note**

This subsection was originally written by Robert A. Duff and published as [Gem #12: Limited Types in Ada 2005](#)⁹⁸.

Consider the following type declaration:

Listing 579: type_defaults.ads

```

1 package Type_Defaults is
2     type Color_Enum is (Red, Blue, Green);
3
4     type T is private;
5 private
6     type T is
7         record
8             Color      : Color_Enum := Red;
9             Is_Gnarly  : Boolean := False;
10            Count      : Natural;
11        end record;

```

(continues on next page)

⁹⁷ <http://www.ada-auth.org/standards/22aarm/html/AA-3-8.html>

⁹⁸ <https://www.adacore.com/gems/ada-gem-12>

(continued from previous page)

```

12
13     procedure Do_Something;
14 end Type_Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_Init
 ↪Init
 MD5: 218154278081f89595534bc02e34539b

If we want to say, "make **Count** equal **100**, but initialize Color and Is_Gnarly to their defaults", we can do this:

Listing 580: type_defaults.adb

```

1 package body Type_Defaults is
2
3     Object_100 : constant T :=
4         (Color      => <>,
5          Is_Gnarly  => <>,
6          Count      => 100);
7
8     procedure Do_Something is null;
9
10 end Type_Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_Init
 ↪Init
 MD5: e64f8881ee74b90dd6058ca8961aae31

Historically

Prior to Ada 2005, the following style was common:

Listing 581: type_defaults.adb

```

1 package body Type_Defaults is
2
3     Object_100 : constant T :=
4         (Color      => Red,
5          Is_Gnarly  => False,
6          Count      => 100);
7
8     procedure Do_Something is null;
9
10 end Type_Defaults;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_Init
 ↪Init
 MD5: c1ddfae75d7f0c691356027903a6d144

Here, we only wanted Object_100 to be a default-initialized T, with **Count** equal to **100**. It's a little bit annoying that we had to write the default values Red and **False** twice. What if we change our mind about Red, and forget to change it in all the relevant places? Since Ada 2005, the <> notation comes to the rescue, as we've just seen.

On the other hand, if we want to say, "make **Count** equal **100**, but initialize all other com-

ponents, including the ones we might add next week, to their defaults", we can do this:

Listing 582: type_defaults.adb

```

1 package body Type_Defaults is
2
3   Object_100 : constant T := (Count => 100,
4                               others => <>);
5
6   procedure Do_Something is null;
7
8 end Type_Defaults;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_Init
 ↪ Init
 MD5: 93f5d71ae80ff0ebad54f2569539f536

Note that if we add a component `Glorp : Integer;` to type `T`, then the `others` case leaves `Glorp` undefined just as this code would do:

Listing 583: type_defaults.adb

```

1 package body Type_Defaults is
2
3   procedure Do_Something is
4     Object_100 : T;
5   begin
6     Object_100.Count := 100;
7   end Do_Something;
8
9 end Type_Defaults;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Default_Init
 ↪ Init
 MD5: 6d328318e2695516794df33466fa5283

Therefore, you should be careful and think twice before using `others`.

Advanced Usages

In addition to expressions such as subprogram calls, we can use *per-object expressions* (page 514) for the default value of a record component. (We discuss this topic later on in more details.)

For example:

Listing 584: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3   type T (D : Positive) is private;
4
5 private
6
7   type T (D : Positive) is record
8     V : Natural := D - 1;
9     --      ^^^^^
10    --      Per-object expression
```

(continues on next page)

(continued from previous page)

```

11     end record;
12
13 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Default_Initialization.Per_Object_Expressions
 MD5: 92591ea482db2b009b8eeafe633ca6cd

In this example, component V is initialized by default with the per-object expression $D - 1$, where D refers to the discriminant D.

25.4.2 Mutually dependent types

In this section, we discuss how to use *incomplete types* (page 310) to declare mutually dependent types. Let's start with this example:

Listing 585: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T1 is record
4         B : T2;
5     end record;
6
7     type T2 is record
8         A : T1;
9     end record;
10
11 end Mutually_Dependent;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_Dependent
 MD5: ffa8d6ab83a1172dcbae0978952dacb2

Build output

```

mutually_dependent.ads:4:11: error: "T2" is undefined
gprbuild: *** compilation phase failed

```

When you try to compile this example, you get a compilation error. The first problem with this code is that, in the declaration of the T1 record, the compiler doesn't know anything about T2. We could solve this by declaring an incomplete type (**type T2;**) before the declaration of T1. This, however, doesn't solve all the problems in the code: the compiler still doesn't know the size of T2, so we cannot create a component of this type. We could, instead, declare an access type and use it here. By doing this, even though the compiler doesn't know the size of T2, it knows the size of an access type designating T2, so the record component can be of such an access type.

To summarize, in order to solve the compilation error above, we need to:

- use at least one incomplete type;
- declare at least one component as an access to an object.

For example, we could declare an incomplete type T2 and then declare the component B of the T1 record as an access to T2. This is the corrected version:

Listing 586: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T2;
4     type T2_Access is access T2;
5
6     type T1 is record
7         B : T2_Access;
8     end record;
9
10    type T2 is record
11        A : T1;
12    end record;
13
14 end Mutually_Dependent;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_↵Dependent
MD5: 1ae10638624a97fa18b9d8f96bfa74ed

We could strive for consistency and declare two incomplete types and two accesses, but this isn't strictly necessary in this case. Here's the adapted code:

Listing 587: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T1;
4     type T1_Access is access T1;
5
6     type T2;
7     type T2_Access is access T2;
8
9     type T1 is record
10        B : T2_Access;
11    end record;
12
13    type T2 is record
14        A : T1_Access;
15    end record;
16
17 end Mutually_Dependent;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Mutually_Dependent_Types.Mutually_↵Dependent
MD5: 9a9899cd0dd2525bd27d67d6629a0071

Later on, we'll see that these code examples can be written using *anonymous access types* (page 1021).

** In the Ada Reference Manual**

- 3.10.1 Incomplete Type Declarations⁹⁹

⁹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html>

25.4.3 Null records

A null record is a record that doesn't have any components. Consequently, it cannot store any information. When declaring a null record, we simply write **null** instead of declaring actual components, as we usually do for records. For example:

Listing 588: null_recs.ads

```

1 package Null_Recs is
2
3     type Null_Record is record
4         null;
5     end record;
6
7 end Null_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
MD5: 3c82da822710342354134fa71a03452a

Note that the syntax can be simplified to **is null record**, which is much more common than the previous form:

Listing 589: null_recs.ads

```

1 package Null_Recs is
2
3     type Null_Record is null record;
4
5 end Null_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
MD5: 1da1746ce5b0a237276272d2b620e282

Although a null record doesn't have components, we can still specify subprograms for it. For example, we could specify an addition operation for it:

Listing 590: null_recs.ads

```

1 package Null_Recs is
2
3     type Null_Record is null record;
4
5     function "+" (A, B : Null_Record)
6         return Null_Record;
7
8 end Null_Recs;
```

Listing 591: null_recs.adb

```

1 package body Null_Recs is
2
3     function "+" (A, B : Null_Record)
4         return Null_Record
5     is
6     pragma Unreferenced (A, B);
7     begin
8         return (null record);
9     end "+";
```

(continues on next page)

(continued from previous page)

```

10
11 end Null_Recs;

```

Listing 592: show_null_rec.adb

```

1 with Null_Recs; use Null_Recs;
2
3 procedure Show_Null_Rec is
4   A, B : Null_Record;
5 begin
6   B := A + A;
7   A := A + B;
8 end Show_Null_Rec;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Null_Record
MD5: 3a1c2fbae75541dfb0b2ff4c14d22039

i In the Ada Reference Manual

- 4.3.1 Record Aggregates¹⁰⁰

Simple Prototyping

A null record doesn't provide much functionality on itself, as we're not storing any information in it. However, it's far from being useless. For example, we can make use of null records to design an API, which we can then use in an application without having to implement the actual functionality of the API. This allows us to design a prototype without having to think about all the implementation details of the API in the first stage.

Consider this example:

Listing 593: devices.ads

```

1 package Devices is
2
3   type Device is private;
4
5   function Create
6     (Active : Boolean)
7     return Device;
8
9   procedure Reset
10    (D : out Device) is null;
11
12   procedure Process
13    (D : in out Device) is null;
14
15   procedure Activate
16    (D : in out Device) is null;
17
18   procedure Deactivate
19    (D : in out Device) is null;
20
21 private

```

(continues on next page)

¹⁰⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html>

(continued from previous page)

```
22
23     type Device is null record;
24
25     function Create (Active : Boolean)
26         return Device is
27         (null record);
28
29 end Devices;
```

Listing 594: show_device.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Devices;      use Devices;
3
4  procedure Show_Device is
5      A : Device;
6  begin
7      Put_Line ("Creating device...");
8      A := Create (Active => True);
9
10     Put_Line ("Processing on device...");
11     Process (A);
12
13     Put_Line ("Deactivating device...");
14     Deactivate (A);
15
16     Put_Line ("Activating device...");
17     Activate (A);
18
19     Put_Line ("Resetting device...");
20     Reset (A);
21 end Show_Device;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
MD5: 7d2fce20ac33607f7081381b307a564a

Runtime output

```
Creating device...
Processing on device...
Deactivating device...
Activating device...
Resetting device...
```

In the `Devices` package, we're declaring the `Device` type and its primitive subprograms: `Create`, `Reset`, `Process`, `Activate` and `Deactivate`. This is the API that we use in our prototype. Note that, although the `Device` type is declared as a private type, it's still defined as a null record in the full view.

In this example, the `Create` function, implemented as an expression function in the private part, simply returns a null record. As expected, this null record returned by `Create` matches the definition of the `Device` type.

All procedures associated with the `Device` type are implemented as null procedures, which means they don't actually have an implementation nor have any effect. We'll discuss this topic [later on in the course](#) (page 796).

In the `Show_Device` procedure — which is an application that implements our prototype —, we declare an object of `Device` type and call all subprograms associated with that type.

Extending the prototype

Because we're either using expression functions or null procedures in the specification of the Devices package, we don't have a package body for it (as there's nothing to be implemented). We could, however, move those user messages from the Show_Devices procedure to a dummy implementation of the Devices package. This is the adapted code:

Listing 595: devices.ads

```

1 package Devices is
2
3     type Device is null record;
4
5     function Create (Active : Boolean)
6         return Device;
7
8     procedure Reset (D : out Device);
9
10    procedure Process (D : in out Device);
11
12    procedure Activate (D : in out Device);
13
14    procedure Deactivate (D : in out Device);
15
16 end Devices;
```

Listing 596: devices.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Devices is
4
5     function Create (Active : Boolean)
6         return Device
7     is
8         pragma Unreferenced (Active);
9     begin
10         Put_Line ("Creating device...");
11         return (null record);
12     end Create;
13
14    procedure Reset (D : out Device)
15    is
16        pragma Unreferenced (D);
17    begin
18        Put_Line ("Processing on device...");
19    end Reset;
20
21    procedure Process (D : in out Device)
22    is
23        pragma Unreferenced (D);
24    begin
25        Put_Line ("Deactivating device...");
26    end Process;
27
28    procedure Activate (D : in out Device)
29    is
30        pragma Unreferenced (D);
31    begin
32        Put_Line ("Activating device...");
33    end Activate;
34
```

(continues on next page)

(continued from previous page)

```
35  procedure Deactivate (D : in out Device)
36  is
37      pragma Unreferenced (D);
38  begin
39      Put_Line ("Resetting device...");
40  end Deactivate;
41
42  end Devices;
```

Listing 597: show_device.adb

```
1  with Devices; use Devices;
2
3  procedure Show_Device is
4      A : Device;
5  begin
6      A := Create (Active => True);
7      Process (A);
8      Deactivate (A);
9      Activate (A);
10     Reset (A);
11 end Show_Device;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
MD5: 1a21b41f3847f6c132ccbc9696ab7689

Runtime output

```
Creating device...
Deactivating device...
Resetting device...
Activating device...
Processing on device...
```

As we changed the specification of the `Devices` package to not use null procedures, we now need a corresponding package body for it. In this package body, we implement the operations on the `Device` type, which actually just display a user message indicating which operation is being called.

Let's focus on this updated version of the `Show_Device` procedure. Now that we've removed all those calls to `Put_Line` from this procedure and just have the calls to operations associated with the `Device` type, it becomes more apparent that, even though `Device` is just a null record, we can design an application with a sequence of various commands operating on it. Also, when we just read the source-code of the `Show_Device` procedure, there's no clear indication that the `Device` type doesn't actually hold any information.

More complex applications

As we've just seen, we can use null records like any other type and create complex prototypes with them. We could, for instance, design an application that makes use of many null records, or even have types that depend on or derive from null records. Let's see a simple example:

Listing 598: many_devices.ads

```
1  package Many_Devices is
2
```

(continues on next page)

(continued from previous page)

```

3  type Device is null record;
4
5  type Device_Config is null record;
6
7  function Create (Config : Device_Config)
8      return Device is
9      (null record);
10
11 type Derived_Device is new Device;
12
13 procedure Process (D : Derived_Device) is null;
14
15 end Many_Devices;

```

Listing 599: show_derived_device.adb

```

1  with Many_Devices; use Many_Devices;
2
3  procedure Show_Derived_Device is
4      A : Device;
5      B : Derived_Device;
6      C : Device_Config;
7  begin
8      A := Create (Config => C);
9      B := Create (Config => C);
10
11      Process (B);
12 end Show_Derived_Device;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Derived_Device
MD5: 757a3def24c8333a27b64943727d8d4e

In this example, the Create function has a null record parameter (of Device_Config type) and returns a null record (of Device type). Also, we derive the Derived_Device type from the Device type. Consequently, Derived_Device is also a null record (since it's derived from a null record). In the Show_Derived_Device procedure, we declare objects of those types (A, B and C) and call primitive subprograms to operate on them.

This example shows that, even though the types we've declared are *just* null records, they can still be used to represent dependencies in our application.

Implementing the API

Let's focus again on the previous example. After we have an initial prototype, we can start implementing some of the functionality needed for the Device type. For example, we can store information about the current activation state in the record:

Listing 600: devices.ads

```

1  package Devices is
2
3      type Device is private;
4
5      function Create (Active : Boolean)
6          return Device;
7
8      procedure Reset (D : out Device);
9

```

(continues on next page)

(continued from previous page)

```

10  procedure Process (D : in out Device);
11
12  procedure Activate (D : in out Device);
13
14  procedure Deactivate (D : in out Device);
15
16  private
17
18  type Device is record
19      Active : Boolean;
20  end record;
21
22  end Devices;

```

Listing 601: devices.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Devices is
4
5      function Create (Active : Boolean)
6          return Device
7      is
8          pragma Unreferenced (Active);
9      begin
10         Put_Line ("Creating device...");
11         return (Active => Active);
12     end Create;
13
14     procedure Reset (D : out Device)
15     is
16         pragma Unreferenced (D);
17     begin
18         Put_Line ("Processing on device...");
19     end Reset;
20
21     procedure Process (D : in out Device)
22     is
23         pragma Unreferenced (D);
24     begin
25         Put_Line ("Deactivating device...");
26     end Process;
27
28     procedure Activate (D : in out Device)
29     is
30     begin
31         Put_Line ("Activating device...");
32         D.Active := True;
33     end Activate;
34
35     procedure Deactivate (D : in out Device)
36     is
37     begin
38         Put_Line ("Resetting device...");
39         D.Active := False;
40     end Deactivate;
41
42 end Devices;

```

Listing 602: show_device.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Devices;      use Devices;
3
4 procedure Show_Device is
5   A : Device;
6 begin
7   A := Create (Active => True);
8   Process (A);
9   Deactivate (A);
10  Activate (A);
11  Reset (A);
12 end Show_Device;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Device
MD5: 348ce0c110b47a6b6fd1c9fe73ef0558

Build output

```

devices.adb:11:25: warning: aspect Unreferenced specified for "Active" [enabled by
↳ default]

```

Runtime output

```

Creating device...
Deactivating device...
Resetting device...
Activating device...
Processing on device...

```

Now, the Device record contains an Active component, which is used in the updated versions of Create, Activate and Deactivate.

Note that we haven't done any change to the implementation of the Show_Device procedure: it's still the same application as before. As we've been hinting in the beginning, using null records makes it easy for us to first create a prototype — as we did in the Show_Device procedure — and postpone the API implementation to a later phase of the project.

Tagged null records

A null record may be tagged, as we can see in this example:

Listing 603: null_recs.ads

```

1 package Null_Recs is
2
3   type Tagged_Null_Record is
4     tagged null record;
5
6   type Abstract_Tagged_Null_Record is
7     abstract tagged null record;
8
9 end Null_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Tagged_Null_Record
MD5: 918572d2c50911b84c80a9c601b75439

As we see in this example, a type can be **tagged**, or even **abstract tagged**. We discuss abstract types later on in the course.

As expected, in addition to deriving from tagged types, we can also extend them. For example:

Listing 604: devices.ads

```
1 package Devices is
2
3   type Device is private;
4
5   function Create (Active : Boolean)
6     return Device;
7
8   type Derived_Device is private;
9
10 private
11
12   type Device is tagged null record;
13
14   function Create (Active : Boolean)
15     return Device is
16     (null record);
17
18   type Derived_Device is new Device with record
19     Active : Boolean;
20   end record;
21
22   function Create (Active : Boolean)
23     return Derived_Device is
24     (Active => Active);
25
26 end Devices;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Null_Records.Extended_Device
MD5: 15e06a5115cbcb131477b5224a6594db

In this example, we derive `Derived_Device` from the `Device` type and extend it with the `Active` component. (Because we have a type extension, we also need to override the `Create` function.)

Since we're now introducing elements from object-oriented programming, we could consider using interfaces instead of null records. We'll discuss this topic later on in the course.

25.4.4 Record discriminants

We introduced the topic of record discriminants in the *Introduction to Ada course* (page 102). Also, in a previous chapter, we mentioned that record types with unconstrained discriminants without defaults are *indefinite types* (page 304).

In this section, we discuss a couple of details about record discriminants that we haven't covered yet. Although the discussion will be restricted to record discriminants, keep in mind that tasks and protected types can also have discriminants. We'll focus on discriminants for tasks and protected types in separate chapters.

In addition, discriminants can be used to write *per-object expressions* (page 511). We discuss this topic later in this chapter.

In the Ada Reference Manual

- 3.7 Discriminants¹⁰¹

Known and unknown discriminant parts

When it comes to discriminants, a type declaration falls into one of the following three categories: it has either no discriminants at all, known discriminants or unknown discriminants.

In order to have no discriminants, a type simply doesn't have a discriminant part in its declaration. For example:

Listing 605: show_discriminants.ads

```

1 package Show_Discriminants is
2
3     type T_No_Discr is private;
4         --      ^^^
5         --      no discriminant part
6
7 private
8
9     type T_No_Discr is null record;
10
11 end Show_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.No_Discriminant_Part
MD5: f6701bd9c46b265753a258a6f99a5c7a

By using parentheses after the type name, we're defining a discriminant part. In this case, the type can either have unknown or known discriminants. For example:

Listing 606: show_discriminants.ads

```

1 package Show_Discriminants is
2
3     type T_Unknown_Discr (<>) is
4         --      ^^
5         --      Unknown discriminant
6         private;
7
8     type T_Known_Discr (D : Integer) is
9         --      ^^^^^^^^^^^
10        --      Known discriminant
11        private;
12
13 private
14
15     type T_Unknown_Discr is
16         null record;
17
18     type T_Known_Discr (D : Integer) is
19         null record;
20
21 end Show_Discriminants;
```

Code block metadata

¹⁰¹ <http://www.ada-auth.org/standards/12rm/html/RM-3-7.html>

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_Parts
MD5: 486edc81b72473e022bb9e56ebaca559

An unknown discriminant part is represented by (<>) in the partial view — this is basically the so-called *box notation* <> (also known as *box compound delimiter*) in parentheses. We discuss unknown discriminant parts and their peculiarities [later on in this chapter](#) (page 488). In this section, we mainly focus on known discriminants.

We've already seen examples of known discriminants in previous chapters. In simple terms, known discriminants are composed by one or more discriminant specifications, which are similar to subprogram parameters, but without parameter modes. In fact, we can think of discriminants as parameters for a type T, but with the goal of defining specific characteristics or constraints when declaring objects of type T.

Discriminant as constant property

We can think of discriminants as constant properties of a type. In fact, if you want to specify a record component C that shouldn't change, declaring it constant isn't allowed in Ada:

Listing 607: constant_properties.ads

```
1 package Constant_Properties is
2
3     type Rec is record
4         C : constant Integer;
5         -- ^^^^^^^
6         -- ERROR: record components
7         -- cannot be constant.
8         V : Integer;
9     end record;
10
11 end Constant_Properties;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: ba189f437348c5892847d067b0bc2e78

Build output

```
constant_properties.ads:4:11: error: constant component not permitted
gprbuild: *** compilation phase failed
```

A simple solution is to use a record discriminant:

Listing 608: constant_properties.ads

```
1 package Constant_Properties is
2
3     type Rec (C : Integer) is
4     record
5         V : Integer;
6     end record;
7
8 end Constant_Properties;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: b638c2fd78761def2b60e9ae7dceb765

A record discriminant can be accessed as a normal component, but it is read-only, so we cannot change it:

Listing 609: show_constant_property.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Constant_Properties;
4  use  Constant_Properties;
5
6  procedure Show_Constant_Property is
7      R : Rec (10);
8  begin
9      Put_Line ("R.C = "
10               & R.C'Image);
11
12     R.C := R.C + 1;
13     -- ERROR: cannot change
14     --          record discriminant
15 end Show_Constant_Property;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Constant_Properties
MD5: 82cde0032f2cb022e690f1175216fd77

Build output

```

show_constant_property.adb:12:05: error: assignment to discriminant not allowed
gprbuild: *** compilation phase failed
```

In this code example, the compilation fails because we cannot change the C discriminant. In this sense, C is a basically a constant component of the R object.

Private types

As we've seen in previous chapters, private types can have discriminants. For example:

Listing 610: private_with_discriminants.ads

```

1  package Private_With_Discriminants is
2
3      type T (L : Positive) is private;
4
5  private
6
7      type Integer_Array is
8          array (Positive range <>) of Integer;
9
10     type T (L : Positive) is
11         record
12             Arr : Integer_Array (1 .. L);
13         end record;
14
15 end Private_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Private_With_
↳ Discriminants
MD5: 8f63443479e31a187a038381d9a32831

Here, discriminant L is used to specify the constraints of the array component Arr. Note that the same discriminant part must appear in both *the partial and the full view* (page 312) of type T.

Object declaration

As we've already seen, we declare objects of a type T with a discriminant D by specifying the actual value of discriminant D. This is called a *discriminant constraint* (page 482). For example:

Listing 611: recs.ads

```
1 package Recs is
2
3     type T (L : Positive;
4             M : Positive) is
5         null record;
6
7 end Recs;
```

Listing 612: show_object_declaration.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Recs;         use Recs;
4
5 procedure Show_Object_Declaration is
6     A : T (L => 5, M => 6);
7     B : T (7, 8);
8 begin
9     Put_Line ("A.L = "
10              & A.L'Image);
11     Put_Line ("A.M = "
12              & A.M'Image);
13     Put_Line ("B.L = "
14              & B.L'Image);
15     Put_Line ("B.M = "
16              & B.M'Image);
17 end Show_Object_Declaration;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Objects_
↳ Discriminants
MD5: 9daae29be9d0f99980ca152a3aca7363

Runtime output

```
A.L = 5
A.M = 6
B.L = 7
B.M = 8
```

As we can see in the declaration of objects A and B, for the discriminant values, we can use a positional ((7, 8)) or named association ((L => 5, M => 6)).

Object size

Discriminants can have an impact on the object size because we can set the discriminant to constraint a component of an *indefinite subtype* (page 304). For example:

Listing 613: recs.ads

```

1 package Recs is
2
3   type Null_Rec (L : Positive;
4                 M : Positive) is
5     private;
6
7   type Rec_Array (L : Positive) is
8     private;
9
10 private
11
12   type Null_Rec (L : Positive;
13                 M : Positive) is
14     null record;
15
16   type Integer_Array is
17     array (Positive range <>) of Integer;
18
19   type Rec_Array (L : Positive) is
20     record
21       Arr : Integer_Array (1 .. L);
22     end record;
23
24 end Recs;
```

Listing 614: show_object_sizes.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Recs;         use Recs;
4
5 procedure Show_Object_Sizes is
6   Null_Rec_A : Null_Rec (1, 2);
7   Null_Rec_B : Null_Rec (5, 6);
8   Rec_Array_A : Rec_Array (10);
9   Rec_Array_B : Rec_Array (20);
10 begin
11   Put_Line ("Null_Rec_A'Size = "
12           & Null_Rec_A'Size'Image);
13   Put_Line ("Null_Rec_B'Size = "
14           & Null_Rec_B'Size'Image);
15   Put_Line ("Rec_Array_A'Size = "
16           & Rec_Array_A'Size'Image);
17   Put_Line ("Rec_Array_B'Size = "
18           & Rec_Array_B'Size'Image);
19 end Show_Object_Sizes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Objects_
 ↪Discriminants_Size
 MD5: 0abbc12286aff9fe428ea585564cf6d4

Build output

```
show_object_sizes.adb:8:04: warning: variable "Rec_Array_A" is read but never
↳assigned [-gnatwv]
show_object_sizes.adb:9:04: warning: variable "Rec_Array_B" is read but never
↳assigned [-gnatwv]
```

Runtime output

```
Null_Rec_A'Size = 64
Null_Rec_B'Size = 64
Rec_Array_A'Size = 352
Rec_Array_B'Size = 672
```

In this example, `Null_Rec_A` and `Null_Rec_B` have the same size because the type is a null record. However, `Rec_Array_A` and `Rec_Array_B` have different sizes because we're setting the `L` discriminant — which we use to constraint the `Arr` array component of the `Rec_Array` type — to 10 and 20, respectively.

Object assignments

As we've just seen, when we set the values for the discriminants of a type in the object declaration, we're constraining the objects. Those constraints are checked at runtime by the [discriminant check](#) (page 809). If the discriminants don't match, the `Constraint_Error` exception is raised.

Let's see an example:

Listing 615: `recs.ads`

```
1 package Recs is
2
3     type T (L : Positive;
4             M : Positive) is
5         null record;
6
7 end Recs;
```

Listing 616: `show_object_assignments.adb`

```
1 with Recs;          use Recs;
2
3 procedure Show_Object_Assignments is
4     A1, A2 : T (5, 6);
5     B      : T (7, 8);
6 begin
7     A1 := A2;        -- OK
8     B  := A1;        -- ERROR!
9 end Show_Object_Assignments;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Object_Assignments
MD5: 199247f1c0575c6845d85fd1911e1cf2
```

Build output

```
show_object_assignments.adb:8:10: warning: incorrect value for discriminant "L"
↳[enabled by default]
show_object_assignments.adb:8:10: warning: Constraint_Error will be raised at run
↳time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_object_assignments.adb:8 discriminant check failed
```

In this example, the `A1 := A2` assignment is accepted because both `A1` and `A2` have the same constraints `((5, 6))`. However, the `B := A1` assignment is not accepted because the discriminant check fails at runtime.

Note that the discriminant check is not performed when we use *mutable subtypes* (page 469) — we discuss this specific kind of subtypes later on.

Discriminant type

In a discriminant specification, the type of the discriminant can only be a discrete subtype or an *access type* (page 891). Other kinds of types — e.g. composite types such as record types — are illegal for discriminants. However, we can always use them indirectly by using access types. (We'll see an example later.)

In addition to that, we can also use a different kind of access types, namely *anonymous access-to-object subtypes* (page 1002). This specific kind of discriminant is called *access discriminant* (page 1011). We discuss this topic in more details in another chapter.

Let's see a code example:

Listing 617: recs.ads

```

1 package Recs is
2
3   type Usage_Mode is (Off,
4                       Simple_Usage,
5                       Advanced_Usage);
6
7   type Priv_Info is private;
8
9   type Priv_Info_Access is access Priv_Info;
10
11  type Proc_Access is
12    access procedure (P : in out Priv_Info);
13
14  type Priv_Rec (Last : Positive;
15                Usage : Usage_Mode;
16                Info : Priv_Info_Access;
17                Proc : Proc_Access) is
18    private;
19
20  private
21
22    type Priv_Info is record
23      A : Positive;
24      B : Positive;
25    end record;
26
27    type Priv_Rec (Last : Positive;
28                  Usage : Usage_Mode;
29                  Info : Priv_Info_Access;
30                  Proc : Proc_Access) is
31      null record;
32
33 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminants_Subtype
MD5: 4ddbc703d8ffcd6dc31e3715df62931a

In this example, we're declaring the `Priv_Rec` type with the following discriminants:

- The Last discriminant of the scalar (i.e. discrete) type **Positive**;
- The Usage discriminant of the enumeration (i.e. discrete) type `Usage_Mode`;
- The Info discriminant of the *access-to-object type* (page 881) `Priv_Info_Access`;
 - We discuss *access-to-object types as discriminant type* (page 891) in another chapter.
- The Proc discriminant of the *access-to-subprogram type* (page 964) `Proc_Access`;
 - We discuss *access-to-subprogram types as discriminant type* (page 970) in another chapter.

As indicated previously, it's illegal to use a private type or a record type as the type of a discriminant. For example:

Listing 618: recs.ads

```
1 package Recs is
2
3     type Priv_Info is private;
4
5     type Priv_Rec (Info : Priv_Info) is
6         private;
7         --      ^^^^^^^^^^^^^^^^^
8         --  ERROR: cannot use private type
9         --      in discriminant.
10
11 private
12
13     type Priv_Info is record
14         A : Positive;
15         B : Positive;
16     end record;
17
18     type Priv_Rec (Info : Priv_Info) is
19         null record;
20
21 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminants_Subtype_Error
MD5: 17f36f0e09cb069d8215b38adbb46541

Build output

```
recs.ads:5:26: error: discriminants must have a discrete or access type
gprbuild: *** compilation phase failed
```

We cannot use the `Priv_Info` directly as a discriminant type because it's a private type. However, as we've just seen in the previous code example, we use it indirectly by using an access type to this private type (see `Priv_Info_Access` in the code example).

Indefinite subtypes as discriminants

As we already implied, we cannot use indefinite subtypes as discriminants. For example, the following code won't compile:

Listing 619: unconstrained_types.ads

```

1 package Unconstrained_Types is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   type Simple_Record (Arr : Integer_Array) is
7     -----
8     -- ERROR: cannot use indefinite type
9     --      in discriminant.
10    record
11      L : Natural := Arr'Length;
12    end record;
13
14 end Unconstrained_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↳ Indefinite_Types_Error
 MD5: f373b401ef1b179fef15cce0d2077286

Build output

```

unconstrained_types.ads:6:30: error: discriminants must have a discrete or access_
↳ type
gprbuild: *** compilation phase failed
```

Integer_Array is a correct type declaration — although the type itself is indefinite after the declaration. However, we cannot use it as the discriminant in the declaration of Simple_Record. We could, however, have a correct declaration by using discriminants as access values:

Listing 620: unconstrained_types.ads

```

1 package Unconstrained_Types is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   type Integer_Array_Access is
7     access Integer_Array;
8
9   type Simple_Record
10     (Arr : Integer_Array_Access) is
11     record
12       L : Natural := Arr'Length;
13     end record;
14
15 end Unconstrained_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Types.Definite_Indefinite_Subtypes.
 ↳ Indefinite_Types_Error
 MD5: dc8193e3684b172e8503e1c5427cf93d

By adding the `Integer_Array_Access` type and using it in `Simple_Record`'s type declaration, we can indirectly use an indefinite type in the declaration of another indefinite type. We discuss this topic later *in another chapter* (page 891).

Default values

We can specify default values for discriminants. Note, however, that we must either specify default values for **all** discriminants of the discriminant part or for none of them. This contrasts with default values for subprogram parameters, where we can *specify default values for just a subset of all parameters of a specific subprogram* (page 770).

As expected, we can override the default values by specifying the values of each discriminant when declaring an object. Let's see a simple example:

Listing 621: `recs.ads`

```
1 package Recs is
2
3     type T (L : Positive := 1;
4             M : Positive := 2) is
5         private;
6
7     private
8
9     type T (L : Positive := 1;
10            M : Positive := 2) is
11         null record;
12
13 end Recs;
```

Listing 622: `show_object_declaration.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Recs;         use Recs;
4
5 procedure Show_Object_Declaration is
6     A : T;
7     B : T (7, 8);
8 begin
9     Put_Line ("A.L = "
10              & A.L'Image);
11     Put_Line ("A.M = "
12              & A.M'Image);
13     Put_Line ("B.L = "
14              & B.L'Image);
15     Put_Line ("B.M = "
16              & B.M'Image);
17 end Show_Object_Declaration;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_
↪ Default_Value
MD5: 33385c4ba4ed9fc90c55990bde0b70cb

Runtime output

```
A.L = 1
A.M = 2
B.L = 7
B.M = 8
```

In this example, object A makes use of the default values for the discriminants of type T, so it has the discriminants (L => 1, M => 2). In the case of object B, we're specifying the values (L => 7, M => 8), which are used instead of the default values.

Note that we cannot set default values for nonlimited tagged types. The same applies to generic formal types. For example:

Listing 623: recs.ads

```

1 package Recs is
2
3   type TT (L : Positive := 1;
4            M : Positive := 2) is
5       -- ^^^^^^^^^^^^^^^^^^
6       -- ERROR: cannot assign default
7       --       in discriminant of
8       --       nonlimited tagged type.
9       tagged private;
10
11   type LTT (L : Positive := 1;
12            M : Positive := 2) is
13       tagged limited private;
14
15 private
16
17   type TT (L : Positive := 1;
18            M : Positive := 2) is
19       tagged null record;
20
21   type LTT (L : Positive := 1;
22            M : Positive := 2) is
23       tagged limited null record;
24
25 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Discriminant_
 ↳Default_Value_Tagged_Type
 MD5: 94b78f032efe49de0b8198083a25d79b

Build output

```

recs.ads:3:29: error: discriminants of nonlimited tagged type cannot have defaults
recs.ads:4:29: error: discriminants of nonlimited tagged type cannot have defaults
gprbuild: *** compilation phase failed
```

As we can see, compilation fails because of the default values for the discriminants of the nonlimited tagged type TT. In the case of the limited tagged type LTT, the default values for the discriminants are legal.

Mutable subtypes

An unconstrained discriminated subtype with defaults is called a mutable subtype, and a variable of such a subtype is called a mutable variable because the discriminants of such a variable can be changed. An important feature of mutable subtypes is that it allows for changing the discriminants of an object via assignments — in this case, no *discriminant check* (page 809) is performed.

Let's see an example:

Listing 624: mutability.ads

```

1 package Mutability is
2
3     type T_Non_Mutable
4       (L : Positive;
5        M : Positive) is
6       null record;
7
8     type T_Mutable
9       (L : Positive := 1;
10        M : Positive := 2) is
11       null record;
12
13 end Mutability;

```

Listing 625: show_mutable_subtype_assignment.adb

```

1 with Mutability; use Mutability;
2
3 procedure Show_Mutable_Subtype_Assignment is
4   NM_1 : T_Non_Mutable (5, 6);
5   NM_2 : T_Non_Mutable (7, 8);
6
7   M_1  : T_Mutable (7, 8);
8   M_2  : T_Mutable;
9 begin
10   NM_2 := NM_1; -- ERROR!
11   M_2  := M_1;  -- OK
12 end Show_Mutable_Subtype_Assignment;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Mutable_Subtype
MD5: ace4470544bdc6efb0dca7039ca33cbc

Build output

```

show_mutable_subtype_assignment.adb:10:12: warning: incorrect value for
↳discriminant "L" [enabled by default]
show_mutable_subtype_assignment.adb:10:12: warning: Constraint_Error will be
↳raised at run time [enabled by default]

```

Runtime output

```

raised CONSTRAINT_ERROR : show_mutable_subtype_assignment.adb:10 discriminant
↳check failed

```

In this example, the `NM_2 := NM_1` assignment fails because both objects are of a non-mutable subtype with different discriminants, so that the discriminant check fails at run-time. However, the `M_2 := M_1` assignment is OK because both objects are mutable variables. In this case, this assignment changes the discriminants of `M_2` from (`L => 1`, `M => 2`) to (`L => 7`, `M => 8`).

Note that assignments of mutable variables may not always work at runtime. For example, if a discriminant of a mutable subtype is used to constraint a component of indefinite subtype, we might see the corresponding checks fail at runtime. For example:

Listing 626: mutability.ads

```

1 package Mutability is
2
3     type T_Mutable_Array (L : Positive := 10) is
4         private;
5
6 private
7
8     type Integer_Array is
9         array (Positive range <>) of Integer;
10
11    type T_Mutable_Array (L : Positive := 10) is
12        record
13            Arr : Integer_Array (1 .. L);
14        end record;
15
16 end Mutability;

```

Listing 627: show_mutable_subtype_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Mutability; use Mutability;
4
5 procedure Show_Mutable_Subtype_Error is
6     A : T_Mutable_Array (10);
7     B : T_Mutable_Array (20);
8 begin
9     Put_Line ("A'Size = "
10              & A'Size'Image);
11     Put_Line ("B'Size = "
12              & B'Size'Image);
13
14     A := B; -- ERROR!
15 end Show_Mutable_Subtype_Error;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Mutable_Subtype_Error
 MD5: 95bc55e7a01c1160dd2f7139778d2d16

Build output

```

show_mutable_subtype_error.adb:7:04: warning: variable "B" is read but never
↳ assigned [-gnatwv]
show_mutable_subtype_error.adb:14:09: warning: incorrect value for discriminant "L
↳ " [enabled by default]
show_mutable_subtype_error.adb:14:09: warning: Constraint_Error will be raised at
↳ run time [enabled by default]
mutability.ads:11:09: warning: creation of "T_Mutable_Array" object may raise
↳ Storage_Error [enabled by default]

```

Runtime output

```

A'Size = 352
B'Size = 672

raised CONSTRAINT_ERROR : show_mutable_subtype_error.adb:14 discriminant check
↳ failed

```

In this case, the assignment `A := B` raises the `Constraint_Error` exception at runtime. Here, the `Arr` component of each object has a different range: `1 .. 10` for object `A` and `1 .. 20` for object `B`. To prevent this situation, we should declare `T_Mutable_Array` as a limited type, so that assignments are not permitted.

Derived types and subtypes

As expected, we may derive types with discriminants or declare subtypes of it. However, there are a couple of details associated with this, which we discuss now.

Subtypes

When declaring a subtype of a type with discriminants, we have the choice to specify the value of the discriminants for the parent type, or specify no discriminants at all:

Listing 628: subtypes_with_discriminants.ads

```
1 package Subtypes_With_Discriminants is
2
3   type T
4     (L : Positive;
5      M : Positive) is
6     null record;
7
8   subtype Sub_T is T;
9   -- Discriminants are not specified:
10  -- taking the ones from T.
11
12  subtype Sub_T_2 is T
13    (L => 3, M => 4);
14  -- Discriminants are specified:
15  -- taking the ones from Sub_T_2
16
17 end Subtypes_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Subtypes
MD5: 6f02c295f295c81fd20d06f7c710994c

For the `Sub_T` subtype declaration in this example, we don't specify values for the parent type's discriminants. For `Sub_T_2`, in contrast, we set the discriminants to `(L => 3, M => 4)`.

When declaring objects of these subtypes, we need to take the constraints into account:

Listing 629: subtypes_with_discriminants.ads

```
1 package Subtypes_With_Discriminants is
2
3   type T
4     (L : Positive;
5      M : Positive) is
6     null record;
7
8   subtype Sub_T is T;
9   -- Discriminants are not specified:
10  -- taking the ones from T.
11
12  subtype Sub_T_2 is T
13    (L => 3, M => 4);
```

(continues on next page)

(continued from previous page)

```

14  -- Discriminants are specified:
15  -- taking the ones from Sub_T_2
16
17  end Subtypes_With_Discriminants;

```

Listing 630: show_subtypes_with_discriminants.adb

```

1  with Subtypes_With_Discriminants;
2  use  Subtypes_With_Discriminants;
3
4  procedure Show_Subtypes_With_Discriminants is
5      A1 : T (1, 2);
6      A2 : T (3, 4);
7      B1 : Sub_T (1, 2);
8      B2 : Sub_T (3, 4);
9      C2 : Sub_T_2;
10
11     -- C1 : Sub_T_2 (1, 2);
12     --      ^^^^
13     -- ERROR: discriminants already
14     --      constrained
15  begin
16      B1 := A1;
17      -- OK: discriminants match
18
19      B2 := A1;
20      -- CONSTRAINT_ERROR!
21
22      B2 := A2;
23      -- OK: discriminants match
24
25      C2 := A1;
26      -- CONSTRAINT_ERROR!
27
28      C2 := A2;
29      -- OK: discriminants match
30  end Show_Subtypes_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Subtypes
MD5: 9a8516d70e7a53ae332e5c5b6df7f04e

Build output

```

show_subtypes_with_discriminants.adb:19:10: warning: incorrect value for
↳discriminant "L" [enabled by default]
show_subtypes_with_discriminants.adb:19:10: warning: Constraint_Error will be
↳raised at run time [enabled by default]
show_subtypes_with_discriminants.adb:25:10: warning: incorrect value for
↳discriminant "L" [enabled by default]
show_subtypes_with_discriminants.adb:25:10: warning: Constraint_Error will be
↳raised at run time [enabled by default]

```

Runtime output

```

raised CONSTRAINT_ERROR : show_subtypes_with_discriminants.adb:19 discriminant
↳check failed

```

For objects of Sub_T subtype, we *have to* specify the value of each discriminant. On the

other hand, for objects of `Sub_T_2` type, we *cannot* specify the constraints because they have already been defined in the subtype's declaration — in this case, they're always set to `(3, 4)`.

When assigning objects of different subtypes, the discriminant check will be performed — as we *mentioned before* (page 464). In this example, the assignments `B2 := A1` and `C2 := A1` fail because the objects have different constraints.

Derived types

The behavior for derived types is very similar to the one we've just described for subtypes. For example:

Listing 631: `derived_with_discriminants.ads`

```
1 package Derived_With_Discriminants is
2
3   type T
4     (L : Positive;
5      M : Positive) is
6     null record;
7
8   type T_Derived is new T;
9   -- Discriminants are not specified:
10  -- taking the ones from T.
11
12  type T_Derived_2 is new T
13    (L => 3, M => 4);
14  -- Discriminants are specified:
15  -- taking the ones from T_Derived_2
16
17 end Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types
MD5: 1e88f787bd9b568e43fc423c121f24f7

For the `T_Derived` type, we reuse the discriminants of the parent type `T`. For the `T_Derived_2` type, we specify a value for each discriminant of `T`.

As you probably notice, this code looks very similar to the code using subtypes. The main difference between using subtypes and derived types is that, as expected, we have to perform a *type conversion* (page 319) in the assignments:

Listing 632: `show_derived_with_discriminants.adb`

```
1 with Derived_With_Discriminants;
2 use Derived_With_Discriminants;
3
4 procedure Show_Derived_With_Discriminants is
5   A1 : T (1, 2);
6   A2 : T (3, 4);
7   B1 : T_Derived (1, 2);
8   B2 : T_Derived (3, 4);
9   C2 : T_Derived_2;
10
11   -- C1 : Sub_T_2 (1, 2);
12   --           ^^^^
13   -- ERROR: discriminants already
14   --         constrained
15 begin
```

(continues on next page)

(continued from previous page)

```

16  B1 := T_Derived (A1);
17  -- OK: discriminants match
18
19  B2 := T_Derived (A1);
20  -- ERROR!
21
22  C2 := T_Derived_2 (A1);
23  -- CONSTRAINT_ERROR!
24
25  C2 := T_Derived_2 (A2);
26  -- OK: discriminants match
27  end Show_Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types
MD5: 1b32807fcd3b343fbf8ab0d0287ca5bb

Build output

```

show_derived_with_discriminants.adb:22:23: warning: incorrect value for
↳discriminant "L" [enabled by default]
show_derived_with_discriminants.adb:22:23: warning: Constraint_Error will be
↳raised at run time [enabled by default]

```

Runtime output

```

raised CONSTRAINT_ERROR : show_derived_with_discriminants.adb:19 discriminant
↳check failed

```

Once again, a discriminant check is performed when assigning objects to ensure that the type discriminants match. In this code example, the assignments `B2 := A1` and `C2 := A1` fail because the objects have different constraints.

Derived types with renamed discriminants

We could rewrite a type declaration such as `type T_Derived is new T` by explicitly declaring the discriminants. We can do that for the previous code example:

Listing 633: derived_with_discriminants.ads

```

1  package Derived_With_Discriminants is
2
3      type T
4          (L : Positive;
5           M : Positive) is
6          null record;
7
8      -- The declaration:
9      --
10     type T_Derived is new T;
11     --
12     is the same as:
13     --
14     type T_Derived
15         (L : Positive;
16          M : Positive) is
17         new T (L => L, M => M);
18
19  end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Same_Discriminants
MD5: 3ee4b3a70e8ab9ba2684c6a2c695f689

We may, however, rename the discriminants instead. For example, we could rename L and M to X and Y. For example:

Listing 634: derived_with_discriminants.ads

```
1 package Derived_With_Discriminants is
2
3     type T
4         (L : Positive;
5          M : Positive) is
6         null record;
7
8     type T_Derived
9         (X : Positive;
10          Y : Positive) is
11         new T (L => X, M => Y);
12
13 end Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Renamed_Discriminants
MD5: ec2954f538fa63b4d3c7c134527be35d

Of course, if we use named association when declaring objects, we have to use the correct discriminant names:

Listing 635: show_derived_with_discriminants.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Derived_With_Discriminants;
4 use Derived_With_Discriminants;
5
6 procedure Show_Derived_With_Discriminants is
7     A : T (L => 1, M => 2);
8     B : T_Derived (X => 3, Y => 4);
9     -- ~~~~~
10    -- Using correct discriminant names
11 begin
12     Put_Line ("A.L = "
13              & A.L'Image);
14     Put_Line ("A.M = "
15              & A.M'Image);
16     Put_Line ("B.X = "
17              & B.X'Image);
18     Put_Line ("B.Y = "
19              & B.Y'Image);
20 end Show_Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Renamed_Discriminants
MD5: cd3ca2d84c8b7d334b152ebab1955a5e

Runtime output

```
A.L = 1
A.M = 2
B.X = 3
B.Y = 4
```

In essence, the discriminants of both parent and derived types are the same: the only difference is that they are accessed by different names. This allows us to convert from a parent type to a derived type:

Listing 636: show_derived_with_discriminants.adb

```
1 with Derived_With_Discriminants;
2 use Derived_With_Discriminants;
3
4 procedure Show_Derived_With_Discriminants is
5   A : T (L => 1, M => 2);
6   B : T_Derived (X => 1, Y => 2);
7 begin
8   B := T_Derived (A); -- OK
9 end Show_Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
 ↳Renamed_Discriminants
 MD5: d685f16bf3a9d64b4c1f182880455ad0

Here, even though objects A and B have discriminants with different names, the assignment `B := T_Derived (A)` is valid.

Derived types with more constrained discriminants

When deriving types with discriminants, we may use a more constrained type for the discriminants of derived type. For example, if the discriminant D of the parent type is of **Integer** type, the corresponding discriminant of the derived type may use a constrained subtype such as **Natural** or **Positive** — because both **Natural** and **Positive** are subtypes of type **Integer**. For example:

Listing 637: derived_with_discriminants.ads

```
1 package Derived_With_Discriminants is
2
3   type T
4     (L : Integer;
5      M : Integer) is
6     null record;
7
8   type T_Derived_2
9     (X : Natural;
10      Y : Positive) is
11     new T (L => X, M => Y);
12
13 end Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_More_
 ↳Constrained_Discriminants
 MD5: 413f04f1f98dde2a6e0df3ee6955da7f

As expected, the constraints of each discriminant's type are taken into account when evaluating the value that is specified for each discriminant:

Listing 638: show_derived_with_discriminants.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Derived_With_Discriminants;
4 use Derived_With_Discriminants;
5
6 procedure Show_Derived_With_Discriminants is
7   A : T (L => -1, M => -2);
8   B : T_Derived_2 (X => 0, Y => 1);
9 begin
10  Put_Line ("A.L = "
11           & A.L'Image);
12  Put_Line ("A.M = "
13           & A.M'Image);
14  Put_Line ("B.X = "
15           & B.X'Image);
16  Put_Line ("B.Y = "
17           & B.Y'Image);
18 end Show_Derived_With_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_More_Constrained_Discriminants
MD5: 508bb7a6eb93005f8f1e5a937b55473c

Runtime output

```
A.L = -1
A.M = -2
B.X = 0
B.Y = 1
```

Here, we can use (L => -1, M => -2) in the declaration of object A because both discriminants are of **Integer** type. However, in the declaration of object B, we can only use values for the discriminants that are in the range of the **Natural** and **Positive** subtypes, respectively. (If you change the code to use negative values instead, a `Constraint_Error` exception is raised at runtime.)

Extending the discriminant part

As we've seen, we can rename discriminants or use more constrained subtypes for discriminants in derived types. We might also want to add a new discriminant to the derived type — in addition to the discriminants of the parent's type. However, this is considered a type extension, as the new discriminant is part of the type definition.

As an example, we may want to add the A discriminant of **Boolean** type to a derived type. For non-tagged types, such a declaration will trigger a compilation error as expected:

Listing 639: derived_with_discriminants.ads

```
1 package Derived_With_Discriminants is
2
3   type T
4     (L : Positive;
5      M : Positive) is
6     null record;
```

(continues on next page)

(continued from previous page)

```

7
8  type T_Derived
9      (X : Positive;
10       Y : Positive;
11       A : Boolean) is
12  --  ^^^^^^^^^^^
13  --  ERROR: cannot extend type with new
14  --  Boolean discriminant A
15      new T (L => X, M => Y);
16
17 end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↳ Extension_Error
MD5: f9ba4ae1c344d63ed706005e30fe60c2

Build output

```

derived_with_discriminants.ads:11:07: error: new discriminants must constrain old_
↳ ones
gprbuild: *** compilation phase failed

```

To circumvent this issue, we could, of course, declare a component of T type instead of deriving from it:

Listing 640: derived_with_discriminants.ads

```

1  package Derived_With_Discriminants is
2
3      type T
4          (L : Positive;
5           M : Positive) is
6          null record;
7
8      type T_2
9          (X : Positive;
10           Y : Positive;
11           A : Boolean) is
12      record
13          A_Comp : T (L => X, M => Y);
14      end record;
15
16 end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_
↳ Extension_Error
MD5: 41e911cdc8486cd49931b2082586d8e7

In this case, A_Comp is a component of type T, and we're using the discriminant X and Y as the constraints of this component.

Naturally, using tagged types is another alternative:

Listing 641: derived_with_discriminants.ads

```

1  package Derived_With_Discriminants is
2

```

(continues on next page)

(continued from previous page)

```

3  type T
4    (L : Positive;
5     M : Positive) is
6    tagged null record;
7
8  type T_Derived_Extended
9    (X : Positive;
10   Y : Positive;
11   A : Boolean) is -- New discriminant
12   new T (L => X, M => Y)
13   with null record;
14
15  type T_Derived_Extended_2
16    (A : Boolean;      -- New discriminant
17   X : Positive;
18   Y : Positive) is
19   new T (L => X, M => Y)
20   with null record;
21
22  type T_Derived_Extended_3
23    (A : Boolean) is -- New discriminant
24   new T (L => 1, M => 2)
25   with null record;
26
27  type T_Derived_Extended_4
28    (A : Boolean;      -- New discriminant
29   X : Positive) is
30   new T (L => X, M => X)
31   with null record;
32
33  end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Tagged_Types
MD5: b8124d132a4b5066826980c8cc43a7ad

In this code example, we're adding the A discriminant when declaring T_Derived_Extended. Because T is a tagged type, such a new discriminant is fine.

Note that the order of the discriminants can be rearranged: when deriving a new type, we don't need to specify the discriminants of the parent type before any new discriminants. In fact, in the declaration of T_Derived_Extended_2, the additional discriminant A is declared before the discriminants that match the parent type's discriminants.

In addition, we may even use literals to specify the constraints for the parent type — as we're doing in the declaration of T_Derived_Extended_3. Also, we can use the same discriminant from the derived type for the constraints of the parent type — in the declaration of T_Derived_Extended_4, we use the X discriminant for both L and M discriminants of type T.

Deriving with defaults

If the discriminants of the parent type have default values, those default values are inherited by the derived type. Alternatively, we can set different default values.

Let's see a code example:

Listing 642: derived_with_discriminants.ads

```

1 package Derived_With_Discriminants is
2
3     type T
4       (L : Positive := 1;
5        M : Positive := 2) is
6       null record;
7
8     type T_Derived is new T;
9
10    type T_Derived_2
11      (L : Positive := 1;
12       M : Positive := 3) is
13      new T (L => L, M => M);
14
15 end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Defaults
 MD5: 4ffa513c0cd8b6812359a2fc4d8325d2

In this example, the derived type `T_Derived` has the same default values as the parent type `T`, namely (`L => 1`, `M => 2`). For the derived type `T_Derived_2`, we're changing the value of `M` to 3 and keeping the same value for `L`.

As we've seen before, instead of setting default values, we can set the constraints of the parent type in the declaration of the derived type:

Listing 643: derived_with_discriminants.ads

```

1 package Derived_With_Discriminants is
2
3     type T
4       (L : Positive := 1;
5        M : Positive := 2) is
6       null record;
7
8     type T_Derived_Constrained is new T
9       (L => 1, M => 3);
10
11 end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Defaults_Constraints
 MD5: d6053fc79a3e7010ec7b3ec73f51f4e5

In this case, we're constraining the discriminants of the parent type to (`L => 1`, `M => 3`). Note that `L` has the same value as the default value set for the parent type `T`.

** For further reading...**

In other contexts (such as *record aggregates* (page 519), which we discuss in another chapter), we could use the so-called *box notation* (page 522) to specify that we want to use the default value. This, however, isn't possible with type discriminants:

Listing 644: derived_with_discriminants.ads

```

1 package Derived_With_Discriminants is
2
3   type T
4     (L : Positive := 1;
5      M : Positive := 2) is
6     null record;
7
8   type T_Derived_Constraint is new T
9     (L => <>, M => 3);
10    ^^^^^^
11    -- ERROR: cannot use default values
12    --      via box notation
13 end Derived_With_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants.Derived_Types_Defaults_Constraints_Box_Notation
MD5: 18d755ec4de45164a47009ab25368452

Build output

```

derived_with_discriminants.ads:9:11: error: missing operand
gprbuild: *** compilation phase failed

```

Instead of using <>, we have to repeat the value explicitly.

25.4.5 Discriminant constraints and operations

In this section, we discuss some details about discriminant constraints and operations related to discriminants — more specifically, the Constrained attribute.

In the Ada Reference Manual

- 3.7.1 Discriminant Constraints¹⁰²

Discriminant constraints

As we discussed before, when *declaring an object with a discriminant* (page 462), we have to specify the values of the all discriminants — unless, of course, those discriminants have a *default value* (page 468). The values we specify for the discriminants are called discriminant constraints.

Let's revisit the code example we've seen earlier on:

Listing 645: recs.ads

```

1 package Recs is
2
3   type T (L : Positive;
4          M : Positive) is
5     null record;
6
7 end Recs;

```

¹⁰² <http://www.ada-auth.org/standards/12rm/html/RM-3-7-1.html>

Listing 646: show_object_declaration.adb

```

1  with Recs;          use Recs;
2
3  procedure Show_Object_Declaration is
4      A : T (L => 5, M => 6);
5      B : T (7, 8);
6      C : T (7, M => 8);
7  begin
8      null;
9  end Show_Object_Declaration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_Operations.Discriminant_Constraint
MD5: 9e37a1cde73f27b99fd2a9eb57f23c44

Here, `L => 5`, `M => 6` (for object A) are named constraints, while `7`, `8` (for object B) are positional constraints.

It's possible to use both positional and named constraints, as we do for object C: `7`, `M => 8`. In this case, the positional associations must precede the named associations.

In the case of named constraints, we can use multiple selector names:

Listing 647: show_object_declaration.adb

```

1  with Recs;          use Recs;
2
3  procedure Show_Object_Declaration is
4      A : T (L | M => 5);
5      --      ^^^^^
6      --  multiple selector names
7  begin
8      null;
9  end Show_Object_Declaration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_Operations.Discriminant_Constraint
MD5: b6fbe1d69bb520a7b6845536a1601978

This is only possible, however, if those named discriminants are all of the same type. (In this case, `L` and `M` are both of **Positive** subtype.)

** In the Ada Reference Manual**

- [3.7.1 Discriminant Constraints](#)¹⁰³

Discriminant constraint in subtypes

We can use discriminant constraints in the declaration of subtypes. For example:

¹⁰³ <http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html>

Listing 648: show_object_declaration.adb

```

1 with Recs;          use Recs;
2
3 procedure Show_Object_Declaration is
4   subtype T_5_6 is T (L => 5, M => 6);
5   --      ~~~~~
6   -- discriminant constraints for subtype
7
8   A : T_5_6;
9 begin
10  null;
11 end Show_Object_Declaration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_Operations.Discriminant_Constraint
MD5: c6c4226073f4282d68d621519ca4d420

In this example, we use the named discriminant constraints `L => 5`, `M => 6` in the declaration of the subtype `T_5_6`.

Constrained Attribute

We can use the Constrained attribute to verify whether an object of discriminated type is constrained or not. Let's look at a simple example:

Listing 649: recs.ads

```

1 package Recs is
2
3   type T (L : Positive := 1) is
4     null record;
5
6 end Recs;

```

Listing 650: show_constrained_attribute.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Recs;          use Recs;
4
5 procedure Show_Constrained_Attribute is
6   Constr  : T (L => 5);
7   --      ^^^^^ constrained.
8   Unconstr : T;
9   --      ^ unconstrained;
10  --      using defaults.
11 begin
12   Put_Line ("Constr'Constrained:  "
13            & Constr'Constrained'Image);
14   Put_Line ("Unconstr'Constrained: "
15            & Unconstr'Constrained'Image);
16 end Show_Constrained_Attribute;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_Operations.Simple_Constrained_Attribute
MD5: 6a9a807f5af132a07949d2887fa5bfe5

Runtime output

```
Constr'Constrained:  TRUE
Unconstr'Constrained: FALSE
```

As the `Constrained` attribute indicates, the `Constr` object is constrained (by the `L => 5` discriminant constraint), while the `Unconstr` object is unconstrained. Note that, even though `Unconstr` is using the default value for `L` — which would correspond to the discriminant constraint `L => 1` — the object itself hasn't been constrained at its declaration.

Let's continue our discussion with a more complex example by reusing the `Unconstrained_Types` package that we declared in a [previous section](#) (page 304). In this version of the package, we're adding a `Reset` procedure for the discriminated record type `Simple_Record`:

Listing 651: `unconstrained_types.ads`

```
1 package Unconstrained_Types is
2
3     type Simple_Record
4       (Extended : Boolean := False) is
5     record
6       V : Integer;
7       case Extended is
8         when False =>
9           null;
10        when True =>
11          V_Float : Float;
12        end case;
13     end record;
14
15     procedure Reset (R : in out Simple_Record);
16
17 end Unconstrained_Types;
```

Listing 652: `unconstrained_types.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Unconstrained_Types is
4
5     procedure Reset (R : in out Simple_Record) is
6       Zero_Not_Extended : constant
7         Simple_Record := (Extended => False,
8                           V      => 0);
9
10      Zero_Extended : constant
11        Simple_Record := (Extended => True,
12                          V      => 0,
13                          V_Float => 0.0);
14    begin
15      Put_Line ("---- Reset: R'Constrained => "
16              & R'Constrained'Image);
17
18      if not R'Constrained then
19        R := Zero_Extended;
20      else
21        if R.Extended then
22          R := Zero_Extended;
23        else
24          R := Zero_Not_Extended;
25        end if;
26    end if;
```

(continues on next page)

(continued from previous page)

```

26     end if;
27     end Reset;
28
29 end Unconstrained_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_Operations.Constrained_Attribute
 MD5: b56e6d71fd4f05e8490412d7fe40b923

As the name indicates, the `Reset` procedure initializes all record components with zero. Note that we use the `Constrained` attribute to verify whether objects are constrained before assigning to them. For objects that are not constrained, we can simply assign another object to it — as we do with the `R := Zero_Extended` statement. When an object is constrained, however, the discriminants must match. If we assign an object to `R`, the discriminant of that object must match the discriminant of `R`. This is the kind of verification that we do in the `else` part of that procedure: we check the state of the `Extended` discriminant before assigning an object to the `R` parameter.

Note that the `Simple_Record` type has a *variant part* (page 498). We discuss this topic later on in this chapter.

Note as well that, in the initialization of the `Zero_Not_Extended` and `Zero_Extended` constants, we have to indicate the discriminant as a component of the aggregates (e.g.: `(Extended => False, V => 0)`). We discuss this topic in another chapter when we learn more about *aggregates and record discriminants* (page 527).

The `Using_Constrained_Attribute` procedure below declares two objects of `Simple_Record` type: `R1` and `R2`. Because the `Simple_Record` type has a default value for its discriminant, we can declare objects of this type without specifying a value for the discriminant. This is exactly what we do in the declaration of `R1`. Here, we don't specify any constraints, so that it takes the default value (`Extended => False`). In the declaration of `R2`, however, we explicitly set `Extended` to `False`:

Listing 653: `using_constrained_attribute.adb`

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2
3  with Unconstrained_Types; use Unconstrained_Types;
4
5  procedure Using_Constrained_Attribute is
6      R1 : Simple_Record;
7      R2 : Simple_Record (Extended => False);
8
9      procedure Show_Rs is
10         begin
11             Put_Line ("R1'Constrained => "
12                     & R1'Constrained'Image);
13             Put_Line ("R1.Extended => "
14                     & R1.Extended'Image);
15             Put_Line ("--");
16             Put_Line ("R2'Constrained => "
17                     & R2'Constrained'Image);
18             Put_Line ("R2.Extended => "
19                     & R2.Extended'Image);
20             Put_Line ("-----");
21         end Show_Rs;
22     begin
23         Show_Rs;
24

```

(continues on next page)

(continued from previous page)

```

25     Reset (R1);
26     Reset (R2);
27     Put_Line ("-----");
28
29     Show_Rs;
30 end Using_Constrained_Attribute;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Records.Discriminants_Constraints_
Operations.Constrained_Attribute
MD5: f7517fcd3c68a784f55064f188d4e7bb
```

Runtime output

```

R1'Constrained => FALSE
R1.Extended => FALSE
--
R2'Constrained => TRUE
R2.Extended => FALSE
-----
---- Reset: R'Constrained => FALSE
---- Reset: R'Constrained => TRUE
-----
R1'Constrained => FALSE
R1.Extended => TRUE
--
R2'Constrained => TRUE
R2.Extended => FALSE
-----
```

When we run this code, the user messages from `Show_Rs` indicate to us that `R1` is not constrained, while `R2` is constrained. Because we declare `R1` without specifying a value for the `Extended` discriminant, `R1` is not constrained. In the declaration of `R2`, on the other hand, the explicit value for the `Extended` discriminant makes this object constrained. Note that, for both `R1` and `R2`, the value of `Extended` is **False** in the declarations.

As we were just discussing, the `Reset` procedure includes checks to avoid mismatches in discriminants. When we don't have those checks, we might get exceptions at runtime. We can force this situation by replacing the implementation of the `Reset` procedure with the following lines:

```

-- [...]
begin
    Put_Line ("---- Reset: R'Constrained => "
              & R'Constrained'Image);
    R := Zero_Extended;
end Reset;
```

Running the code now generates a runtime exception:

```
raised CONSTRAINT_ERROR : unconstrained_types.adb:12 discriminant check failed
```

This exception is raised during the call to `Reset (R2)`. As we see in the code, `R2` is constrained. Also, its `Extended` discriminant is set to **False**, which means that it doesn't have the `V_Float` component. Therefore, `R2` is not compatible with the constant `Zero_Extended` object, so we cannot assign `Zero_Extended` to `R2`. Also, because `R2` is constrained, its `Extended` discriminant cannot be modified.

The behavior is different for the call to `Reset (R1)`, which works fine. Here, when we pass `R1` as an argument to the `Reset` procedure, its `Extended` discriminant is **False** by

default. Thus, R1 is also not compatible with the Zero_Extended object. However, because R1 is not constrained, the assignment modifies R1 (by changing the value of the Extended discriminant). Therefore, with the call to Reset, the Extended discriminant of R1 changes from **False** to **True**.

In the Ada Reference Manual

- 3.7.2 Operations of Discriminated Types¹⁰⁴

25.4.6 Unknown discriminants

As we've seen *previously* (page 459), a type with discriminants can have known discriminants or unknown discriminants. In this section, we focus on unknown discriminants. Because the discriminants are unknown, this is an *indefinite type* (page 304). Let's start with a simple example:

Listing 654: unknown_discriminants.ads

```
1 package Unknown_Discriminants is
2
3     type T_Unknown_Discr (<=>) is
4         --      ^^^^
5         --      Unknown discriminant part
6         private;
7
8 private
9
10    type T_Unknown_Discr is
11        null record;
12
13 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Simple_Example
MD5: 5f673c957132b1bca633c247f857e37b

Note that we can only use an unknown discriminant part in the *partial view* (page 312); we cannot use it in the full view of a type:

Listing 655: unknown_discriminants.ads

```
1 package Unknown_Discriminants is
2
3     type T_Unknown_Discr (<=>) is
4         null record;
5
6 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Wrong_Full_View
MD5: dfce1471556af87b6a99314b1ee32446

Build output

¹⁰⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-7-2.html>

```
unknown_discriminants.ads:3:25: error: full type declaration cannot have unknown_
↳ discriminants
gprbuild: *** compilation phase failed
```

To be more precise, an unknown discriminant part can only be used in the declaration of a private type, a private extension or an *incomplete type* (page 310). In addition, as we'll see in another chapter, it can also be used in the generic equivalents: generic private types, generic private extensions, generic incomplete types, and formal derived types.

For example:

Listing 656: unknown_discriminants.ads

```
1 package Unknown_Discriminants is
2
3   -- Private type
4   type Rec (<>) is
5     private;
6
7   -- Tagged private type
8   type Tagged_Rec (<>) is
9     tagged private;
10
11  -- Incomplete type
12  type T_Incomplete (<>);
13
14  type T_Incomplete (<>) is
15    private;
16
17 private
18
19  type Rec is
20    null record;
21
22  type Tagged_Rec is
23    tagged null record;
24
25  type T_Incomplete is
26    null record;
27
28 end Unknown_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Simple_
↳ Example
MD5: e601ab326c43501e36e0d4656dc1629e
```

In this example, we have three forms of private types using an unknown discriminant part: an untagged private type (Rec), a tagged type (Tagged_Rec) and an incomplete type (T_Incomplete) that becomes an untagged private type.

In the Ada Reference Manual

- 3.7 Discriminants¹⁰⁵

¹⁰⁵ <http://www.ada-auth.org/standards/12rm/html/RM-3-7.html>

Object declaration

Now, let's talk about objects of types with unknown discriminants. Consider the `Rec` type below:

Listing 657: `unknown_discriminants.ads`

```
1 package Unknown_Discriminants is
2
3     type Rec (<>) is private;
4
5 private
6
7     type Rec is
8     record
9         I : Integer;
10    end record;
11
12 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Object_Declaration
MD5: 9f588870ec70ea30c795a6a0a602f589

We cannot declare objects of type `Rec` *directly*, as this type is *indefinite* (page 304):

Listing 658: `show_object_declaration.adb`

```
1 with Unknown_Discriminants;
2 use Unknown_Discriminants;
3
4 procedure Show_Object_Declaration is
5     A : Rec;
6 begin
7     null;
8 end Show_Object_Declaration;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Object_Declaration
MD5: 5f30773fc17096943939468faf50338b

Build output

```
show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need ↵
↵ initialization)
gprbuild: *** compilation phase failed
```

Because the type is indefinite, it requires explicit initialization — we can do this by introducing a subprogram that initializes the type. In our code example, we can implement a simple `Init` function for this type:

Listing 659: `unknown_discriminants.ads`

```
1 package Unknown_Discriminants is
2
3     type Rec (<>) is private;
4
5     function Init return Rec;
```

(continues on next page)

(continued from previous page)

```

6
7 private
8
9     type Rec is
10    record
11        I : Integer;
12    end record;
13
14    function Init return Rec is
15        ((I => 0));
16
17 end Unknown_Discriminants;

```

Listing 660: show_constructor_function.adb

```

1 with Unknown_Discriminants;
2 use Unknown_Discriminants;
3
4 procedure Show_Constructor_Function is
5     R : Rec := Init;
6 begin
7     null;
8 end Show_Constructor_Function;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳ Object_Declaration
 MD5: 1cee0c4b883b3a0c25fae0a5111db2a8

In the `Show_Constructor_Function` procedure from this example, we call the `Init` function to initialize the `R` object in its declaration (of `Rec` type). Note that for this specific type, this is the only possible way to declare the `R` object. In fact, compilation fails if we write `R : Rec;`.

Using a private type with unknown discriminants is an important Ada idiom, as we gain extra control over its initialization. For example, if we have to ensure that certain components of the private record are initialized when an object is being declared, we can perform this initialization in the `Init` function — instead of just hoping that an initialization function is called for this object at some point. Also, if further information is needed to initialize an object, we can add parameters to the `Init` function, thereby forcing the user to provide this information.

For even more control over objects, we can use *limited types with unknown discriminants* (page 1096).

Partial and full view

As we've just seen, if we declare a type with an unknown discriminant part, we can only use it in the partial view. In the full view, we cannot use an unknown discriminant part, but have to use either no discriminants or known discriminants. For example:

Listing 661: unknown_discriminants.ads

```

1 package Unknown_Discriminants is
2
3     type Rec_No_Discr (<>) is private;
4
5     type Rec_Known_Discr (<>) is private;
6

```

(continues on next page)

(continued from previous page)

```
7 private
8
9     type Rec_No_Discr is null record;
10
11     type Rec_Known_Discr
12       (L : Positive) is null record;
13
14 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Partial_Full_View
MD5: 3d37dcc9d1b12bf9a189cf515b168430

In this example, `Rec_No_Discr` has no discriminants in its full view, while `Rec_Known_Discr` has the discriminant `L`.

In addition, the full view can be an (unconstrained) array type as well:

Listing 662: `unknown_discriminants.ads`

```
1 package Unknown_Discriminants is
2
3     type Arr (<>) is private;
4
5 private
6
7     type Arr is
8       array (Positive range <>)
9         of Integer;
10
11 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Partial_Full_View
MD5: d1e0f60048c6ca6bcf863a8c0cf68314

Here, the full view of `Arr` is an array type.

In the Ada Reference Manual

- [3.7 Discriminants](#)¹⁰⁶

Derived types

As expected, we can derive from types with unknown discriminants. Consider the following package:

Listing 663: `unknown_discriminants.ads`

```
1 package Unknown_Discriminants is
2
3     type Rec (<>) is private;
4
```

(continues on next page)

¹⁰⁶ <http://www.ada-auth.org/standards/12rm/html/RM-3-7.html>

(continued from previous page)

```

5 private
6
7     type Rec is null record;
8
9 end Unknown_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 948e7c7ecd00915fa23a98cbaf2bbcbe

We can then declare the Derived_Rec type:

Listing 664: unknown_discriminants-children.ads

```

1 package Unknown_Discriminants.Children is
2
3     type Derived_Rec is
4         new Rec;
5
6 end Unknown_Discriminants.Children;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 1fa7e905c794d48bf6c76ff51e1abd8d

Note that Derived_Rec has unknown discriminants, even though we're not explicitly using an unknown discriminant part ((<>)) in its declaration. (In fact, we're not allowed to use an unknown discriminant part in this case.) Therefore, declaring objects of this type directly isn't possible, just like the parent type Rec:

Listing 665: show_object_declaration.adb

```

1 with Unknown_Discriminants.Children;
2 use Unknown_Discriminants.Children;
3
4 procedure Show_Object_Declaration is
5     A : Derived_Rec;
6 begin
7     null;
8 end Show_Object_Declaration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 5d0b8980e6f60595b9de8a2ea8fa2132

Build output

```

show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need_
↳ initialization)
gprbuild: *** compilation phase failed

```

Deriving from tagged types

We can also derive from tagged types with unknown discriminants. Consider the following package:

Listing 666: unknown_discriminants.ads

```
1 package Unknown_Discriminants is
2
3     type Rec (<>) is tagged private;
4
5 private
6
7     type Rec is tagged null record;
8
9 end Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↳ Tagged_Type
MD5: ef66d098df1c93495bf5f6c6ac86f203

We can derive from the Rec type. In this case, however, we can use an unknown discriminant part, a known discriminant part, or no discriminants:

Listing 667: unknown_discriminants-children.ads

```
1 package Unknown_Discriminants.Children is
2
3     type Derived_Rec_Unknown_Discr (<>) is
4         new Rec with private;
5
6     type Derived_Rec_Known_Discr (L : Positive) is
7         new Rec with private;
8
9     type Derived_Rec_No_Discr is
10         new Rec with private;
11
12 private
13
14     type Derived_Rec_Unknown_Discr is
15         new Rec with null record;
16
17     type Derived_Rec_Known_Discr (L : Positive) is
18         new Rec with null record;
19
20     type Derived_Rec_No_Discr is
21         new Rec with null record;
22
23 end Unknown_Discriminants.Children;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
↳ Tagged_Type
MD5: 98583f0b39c6f8bea49d1781844bb33e

In this example, we declare `Derived_Rec_Unknown_Discr` with an unknown discriminant part, `Derived_Rec_Known_Discr` with a known discriminant part, and `Derived_Rec_No_Discr` with no discriminants.

As expected, `Derived_Rec_Unknown_Discr` has unknown discriminants because it has an unknown discriminant part. In the case of `Derived_Rec_No_Discr`, which has no discriminants, we're deriving the unknown discriminants of `Rec`, so it also has unknown discriminants. In contrast, because `Derived_Rec_Known_Discr` has a known discriminant part, those discriminants are overriding the unknown discriminants of the parent type `Rec`.

Therefore, we can declare objects of `Derived_Rec_Known_Discr` type without explicit initialization:

Listing 668: `show_object_declaration.adb`

```

1 with Unknown_Discriminants.Children;
2 use Unknown_Discriminants.Children;
3
4 procedure Show_Object_Declaration is
5   A : Derived_Rec_Unknown_Discr;
6   -- ERROR: unknown discriminants
7   -- because of the type's
8   -- unknown discriminant part
9
10  B : Derived_Rec_Known_Discr (1);
11  -- OK: known discriminants
12
13  C : Derived_Rec_No_Discr;
14  -- ERROR: unknown discriminants
15  -- because of parent type's
16  -- unknown discriminant part
17 begin
18   null;
19 end Show_Object_Declaration;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
 ↳ Tagged_Type
 MD5: 91f6ae2abf88976833d0e4eff02d4c40

Build output

```

show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need_
↳ initialization)
show_object_declaration.adb:13:08: error: unconstrained subtype not allowed (need_
↳ initialization)
gprbuild: *** compilation phase failed
```

As we can see, we can only directly declare objects of type `Derived_Rec_Known_Discr` because it has known discriminants, while the other two derived types have unknown discriminants — which are explicitly specified (`Derived_Rec_Unknown_Discr`) or implicitly derived from the parent (`Derived_Rec_No_Discr`).

Note that the parent type `Rec` had a requirement for explicit initialization. By using known discriminants in the declaration of `Derived_Rec_Known_Discr`, we're removing this requirement for the derived type.

The contrary is also true: we can derive a type with known discriminants and use an unknown discriminant part:

Listing 669: `unknown_discriminants-children-grand.ads`

```

1 package Unknown_Discriminants.Children.Grand is
2
3   type Grand_Rec_Unknown_Discr (<>) is
4     new Derived_Rec_Known_Discr (1)
5     with private;
6
7 private
8
9   type Grand_Rec_Unknown_Discr is
10    new Derived_Rec_Known_Discr (1)
```

(continues on next page)

(continued from previous page)

```

11     with null record;
12
13 end Unknown_Discriminants.Children.Grand;

```

Listing 670: show_object_declaration.adb

```

1  with Unknown_Discriminants.Children.Grand;
2  use  Unknown_Discriminants.Children.Grand;
3
4  procedure Show_Object_Declaration is
5      A : Grand_Rec_Unknown_Discr;
6      --  ERROR: unknown discriminants
7      --      because of the type's
8      --      unknown discriminant part
9  begin
10     null;
11 end Show_Object_Declaration;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_
 ↳ Tagged_Type
 MD5: 0e931df8294cee1f49b187c43614aa20

Build output

```

show_object_declaration.adb:5:08: error: unconstrained subtype not allowed (need_
↳ initialization)
show_object_declaration.adb:5:08: error: provide initial value or explicit_
↳ discriminant values
show_object_declaration.adb:5:08: error: or give default discriminant values for_
↳ type "Grand_Rec_Unknown_Discr"
gprbuild: *** compilation phase failed

```

In this example, Grand_Rec_Unknown_Discr has unknown discriminants and requires explicit initialization, even though its parent type Derived_Rec_Known_Discr has known discriminants.

** In the Ada Reference Manual**

- 3.7 Discriminants¹⁰⁷

25.4.7 Unconstrained subtypes

A subtype is called an unconstrained subtype if its type has unknown discriminants. Consider a simple Rec type:

Listing 671: unknown_discriminants.ads

```

1  package Unknown_Discriminants is
2
3      type Rec (<>) is private;
4
5  private
6
7      type Rec is null record;

```

(continues on next page)

¹⁰⁷ <http://www.ada-auth.org/standards/12rm/html/RM-3-7.html>

(continued from previous page)

```

8
9 end Unknown_Discriminants;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.
↳Unconstrained_Subtype
MD5: 948e7c7ecd00915fa23a98cbaf2bbcbce

```

A subtype of Rec type is unconstrained:

Listing 672: unknown_discriminants-children.ads

```

1 package Unknown_Discriminants.Children is
2
3     subtype Rec_Unconstrained is Rec;
4
5 end Unknown_Discriminants.Children;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.
↳Unconstrained_Subtype
MD5: 6b76b6a94d8c9487dbeea3256d5de01f

```

In this example, Rec_Unconstrained is an unconstrained subtype because it's derived from the Rec type. We can verify this by triggering a compilation error:

Listing 673: show_object_declaration.adb

```

1 with Unknown_Discriminants.Children;
2 use Unknown_Discriminants.Children;
3
4 procedure Show_Object_Declaration is
5     A : Rec_Unconstrained;
6 begin
7     null;
8 end Show_Object_Declaration;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Derived_Type
MD5: 442fab4d174de31f27d0de56bf9b8422

```

Build output

```

show_object_declaration.adb:5:08: error: "Rec_Unconstrained" is undefined
gprbuild: *** compilation phase failed

```

In addition, if we declare a subtype based on a type that allows range, index, or discriminant constraints, but we don't constraint the subtype, this subtype is also considered an unconstrained subtype. For example:

Listing 674: unconstrained_subtypes.ads

```

1 package Unconstrained_Subtypes is
2
3     type Arr is
4         array (Positive range <>) of
5             Integer;
6

```

(continues on next page)

(continued from previous page)

```

7  type Rec (L : Positive) is
8      null record;
9
10  subtype Arr_Sub is Arr;
11      --
12      -- no constraints
13
14  subtype Rec_Sub is Rec;
15      --
16      -- no constraints
17
18  end Unconstrained_Subtypes;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Unknown_Discriminants.Other_
 ↳ Unconstrained_Subtypes
 MD5: 3ebc2eb371472dc76eb543b4633e59b3

In this example, Arr_Sub and Rec_Sub are unconstrained subtypes.

i In the Ada Reference Manual

- 3.2 Types and Subtypes¹⁰⁸

25.4.8 Variant parts

We've introduced variant records back in the *Introduction to Ada course* (page 104). In simple terms, a variant record is a record with discriminants that allows for varying its structure. Basically, it's a record containing a **case** statement that specifies which record components exist for each discriminant value. For example:

Listing 675: devices.ads

```

1  package Devices is
2
3      type Device_State is
4          (Off, On);
5
6      type Device_Info is
7          record
8              V : Float;
9          end record;
10
11     type Device (State : Device_State := Off) is
12         record
13             case State is
14                 when Off =>
15                     null;
16                 when On =>
17                     Info : Device_Info;
18             end case;
19         end record;
20
21 end Devices;

```

Code block metadata

¹⁰⁸ <http://www.ada-auth.org/standards/12rm/html/RM-3-2.html>

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Simple_Device
 MD5: 3b63a63aef1d9cb00be870c831829158

The Device type from this example has a variant part. Depending on the value of the State discriminant, it can be either a null record (when State is Off) or have the Info component (when State is On).

Let's look at a test application for the Devices package:

Listing 676: show_device.adb

```

1 with Devices; use Devices;
2
3 procedure Show_Device is
4   D      : Device;
5   D_Off  : Device (Off);
6   D_On   : Device (On);
7 begin
8   D := D_Off;
9   -- OK!
10
11  D := D_On;
12  -- OK!
13
14  D_Off := D_On;
15  --      ^^^^
16  --  CONSTRAINT_ERROR!
17
18  D_On := D_Off;
19  --      ^^^^
20  --  CONSTRAINT_ERROR!
21 end Show_Device;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Simple_Device
 MD5: a11e2739131f435e8428a5e2a9a478e7

Build output

```

show_device.adb:11:09: warning: "D_On" may be referenced before it has a value.
↳[enabled by default]
show_device.adb:14:13: warning: incorrect value for discriminant "State" [enabled
↳by default]
show_device.adb:14:13: warning: Constraint_Error will be raised at run time.
↳[enabled by default]
show_device.adb:18:13: warning: incorrect value for discriminant "State" [enabled
↳by default]
show_device.adb:18:13: warning: Constraint_Error will be raised at run time.
↳[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_device.adb:14 discriminant check failed
```

As we've discussed [previously](#) (page 464), when we set the values for the discriminants of a type in the object declaration, we're constraining the objects. If the discriminants of two objects don't match, the Constraint_Error exception is raised at runtime because the [discriminant check](#) (page 809) fails. Therefore, in the Show_Device procedure, because D_Off and D_On are constrained and have different values for the State discriminant, we cannot assign them to each other. In contrast, because D wasn't constrained at its declaration, we

can assign objects with different discriminants (such as `D_Off` and `D_On`) to it.

Note that the variant part of a record can be more complex. For example, we could have an additional discriminant and use it in the variant part:

Listing 677: `devices.ads`

```
1 package Devices is
2
3   type Device_State is
4     (Off, On);
5
6   type Device_Info is
7     record
8       V : Float;
9     end record;
10
11   type Device (State : Device_State;
12               Boost : Boolean) is
13     record
14       case State is
15         when Off =>
16           null;
17         when On =>
18           Info : Device_Info;
19           case Boost is
20             when False =>
21               null;
22             when True =>
23               Factor : Float;
24           end case;
25         end case;
26       end record;
27
28 end Devices;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device_Boost`
MD5: `4c5e84ccebca9e4ef5e2d6d131ba0e6a`

In this version of the `Devices` package, we introduced a *boost button* as a discriminant (`Boost`) and an associated boost factor component (`Factor`) in the variant part.

In the remaining parts of this section, we discuss a couple of details about variant records.

In the Ada Reference Manual

- [3.8.1 Variant Parts and Discrete Choices](#)¹⁰⁹

Discriminant type and value coverage

The subtype of discriminants used in the variant part must be of a discrete type — it cannot be of an access or a floating-point type, for example. Also, all possible values of the subtype of each discriminant must be covered in the case statement of the variant part. For example, consider the following variant record:

¹⁰⁹ <http://www.ada-auth.org/standards/12rm/html/RM-3-8-1.html>

Listing 678: subtype_coverage.ads

```

1 package Subtype_Coverage is
2
3     type Var_Rec (Value : Integer) is
4     record
5         case Value is
6             when 0 .. 100 =>
7                 I : Integer;
8
9             -- ERROR: missing values!
10            end case;
11        end record;
12
13    end Subtype_Coverage;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 084a468bc8d6f63d21f804c9ddc70622

Build output

```

subtype_coverage.ads:5:07: error: missing case values: -16#8000_0000# .. -1
subtype_coverage.ads:5:07: error: missing case values: 101 .. 16#7FFF_FFFF#
gprbuild: *** compilation phase failed
```

This package cannot be compiled because, in the variant part, we're only covering values for the Value discriminant in the range between 0 and 100. To fix this compilation error, we have to cover all values instead. For example:

Listing 679: subtype_coverage.ads

```

1 package Subtype_Coverage is
2
3     type Var_Rec (Value : Integer) is
4     record
5         case Value is
6             when Integer'First .. -1 =>
7                 null;
8             when 0 .. 100 =>
9                 I : Integer;
10            when 101 .. Integer'Last =>
11                null;
12            end case;
13        end record;
14
15    end Subtype_Coverage;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 9dfa0dfc3d3e11394a79b1ab6b61bafc

Of course, specifying all possible values can be difficult. As an alternative, we could simplify the case statement by just using **others** as a discrete choice that encompasses all values that haven't been specified earlier in the case statement:

Listing 680: subtype_coverage.ads

```

1 package Subtype_Coverage is
2
3   type Var_Rec (Value : Integer) is
4     record
5       case Value is
6         when 0 .. 100 =>
7           I : Integer;
8         when others =>
9           null;
10        end case;
11      end record;
12
13 end Subtype_Coverage;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Coverage
MD5: 0b28038d5137de702cb5b8e875fedefa

By using **when others => ...** in this last example, we ensure that all values have been covered.

Record size

When declaring an object, the values we select for the discriminants related to the variant part have an impact on the overall size of that object — in fact, it may be smaller or bigger depending on this selection. Let's see an example:

Listing 681: variant_records.ads

```

1 package Variant_Records is
2
3   type Simple_Record
4     (Extended : Boolean := False) is
5     record
6       V : Integer;
7       case Extended is
8         when False =>
9           null;
10        when True =>
11          V_Float : Float;
12        end case;
13      end record;
14
15 end Variant_Records;
```

Listing 682: show_variant_rec_size.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2
3 with Variant_Records; use Variant_Records;
4
5 procedure Show_Variant_Rec_Size is
6   SR_No_Ext : Simple_Record
7     (Extended => False);
8   SR_Ext    : Simple_Record
9     (Extended => True);
10  SR        : Simple_Record;
```

(continues on next page)

(continued from previous page)

```

11 begin
12     Put_Line ("SR_No_Ext'Size : "
13             & SR_No_Ext'Size'Image
14             & " bits");
15     Put_Line ("SR_Ext'Size : "
16             & SR_Ext'Size'Image
17             & " bits");
18     Put_Line ("SR'Size : "
19             & SR'Size'Image
20             & " bits");
21 end Show_Variant_Rec_Size;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Size
MD5: 4aaf10924e7469d000cefeb70a69a2fa

Build output

```

show_variant_rec_size.adb:6:04: warning: variable "SR_No_Ext" is read but never
↳ assigned [-gnatwv]
show_variant_rec_size.adb:8:04: warning: variable "SR_Ext" is read but never
↳ assigned [-gnatwv]
show_variant_rec_size.adb:10:04: warning: variable "SR" is read but never assigned
↳ [-gnatwv]

```

Runtime output

```

SR_No_Ext'Size : 64 bits
SR_Ext'Size : 96 bits
SR'Size : 96 bits

```

As we can confirm when we run this application, the choice for the discriminant has an impact on the size of the object. In the case of the `SR_No_Ext` object, setting the Extended discriminant to **False** excludes the `V_Float` component. For the `SR_Ext` object, on the other hand, we include the `V_Float` component. Therefore, on a typical PC, the size of `SR_No_Ext` is 8 bytes (4 bytes for the Extended discriminant and 4 bytes for the `V` component), while the size of `SR_Ext` is 12 bytes (i.e., additional 4 bytes for the `V_Float` component).

In the case of `SR`, because the object isn't constrained, the size of the object is 12 bytes on a typical PC — the same size as `SR_Ext`. This is because `SR` has to account for the case when all components must be available, even though the Extended discriminant is set to **False** by default. Remember that an assignment such as `SR := SR_Ext` is valid, so enough memory must be available to ensure that the assignment is performed correctly.

This principle applies to more complicated variant records. For example:

Listing 683: variant_records.ads

```

1 package Variant_Records is
2
3     type Simple_Record
4     (Extended : Boolean := False;
5      Extended_2 : Boolean := False) is
6     record
7         V : Integer;
8         case Extended is
9             when False =>
10                 case Extended_2 is
11                     when False =>
12                         null;

```

(continues on next page)

(continued from previous page)

```

13         when True =>
14             V_Int_2 : Integer;
15             V_Int_3 : Integer;
16         end case;
17     when True =>
18         V_Float : Float;
19         case Extended_2 is
20             when False =>
21                 null;
22             when True =>
23                 V_Float_2 : Float;
24         end case;
25     end case;
26 end record;
27
28 end Variant_Records;

```

Listing 684: show_variant_rec_size.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Variant_Records; use Variant_Records;
4
5  procedure Show_Variant_Rec_Size is
6      SR : Simple_Record;
7  begin
8      Put_Line ("SR'Size : "
9                & SR'Size'Image
10               & " bits");
11 end Show_Variant_Rec_Size;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Size
MD5: 5f0ac936a5fee50cbe88a7b863a1a550

Build output

```
show_variant_rec_size.adb:6:04: warning: variable "SR" is read but never assigned_
↪ [-gnatwv]
```

Runtime output

```
SR'Size : 128 bits
```

In this example, the size of SR is 16 bytes on a typical PC. This accounts for 4 bytes for the discriminants `Extended` and `Extended_2`, and 4 bytes for each of the 3 components that are being taken into account for the worst case:

- components `V`, `V_Int_2` and `V_Int_3` when we set `Extended => False`, `Extended_2 => True`;
- components `V`, `V_Float` and `V_Float_2` when we set `Extended => True`, `Extended_2 => True`.

Note that a memory block is shared between the `V_Int_2` and `V_Int_3` components from the first worst case, and `V_Float` and the `V_Float_2` components from the second worst case. As we can see, the compiler will typically optimize the size of a record as much as possible by assessing which components are really needed for the worst case.

Also, as we discussed previously, we can use *unchecked unions* (page 391) in combination with variant records, which has an impact on the object size.

Ensuring valid information

We can use variant parts to prevent invalid information from being used. Let's look again at the Device type from the previous code example:

```
type Device (State : Device_State) is
record
  case State is
    when Off =>
      null;
    when On =>
      Info : Device_Info;
  end case;
end record;
```

For the sake of this example, we could say that a device that is turned off doesn't have any valuable information. Therefore, the device information stored in the Info component of the Device type is only valid if the device is turned on. Thus, if the device is turned off (i.e., Device_State = Off), we should prevent the application from processing device information that is probably incorrect. Let's extend the previous code example to accommodate this requirement:

Listing 685: devices.ads

```
1 package Devices is
2
3   type Device_State is
4     (Off, On);
5
6   type Device
7     (State : Device_State := Off) is
8     private;
9
10  procedure Turn_Off (D : in out Device);
11
12  procedure Turn_On (D : in out Device);
13
14  type Device_Info is
15  record
16    V : Float;
17  end record;
18
19  function Current_Info (D : Device)
20    return Device_Info;
21
22 private
23
24  type Device (State : Device_State := Off) is
25  record
26    case State is
27      when Off =>
28        null;
29      when On =>
30        Info : Device_Info;
31    end case;
32  end record;
33
34  Device_Off : constant Device :=
35    (State => Off);
36
37  Device_On : constant Device :=
38    (State => On,
```

(continues on next page)

(continued from previous page)

```
39         others => <>);
40
41 end Devices;
```

Listing 686: devices.adb

```
1 package body Devices is
2
3     procedure Turn_Off (D : in out Device) is
4     begin
5         D := Device_Off;
6     end Turn_Off;
7
8     procedure Turn_On (D : in out Device) is
9     begin
10        D := Device_On;
11    end Turn_On;
12
13    function Current_Info (D : Device)
14        return Device_Info is
15        (D.Info);
16
17 end Devices;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: e03db406de3550865dd99986d2c71145

Let's create a test application called Show_Device that makes use of this device by turning it on and off, and by retrieving information from it:

Listing 687: show_device.adb

```
1 with Devices; use Devices;
2
3 procedure Show_Device is
4     D : Device;
5     I : Device_Info;
6 begin
7     Turn_On (D);
8     I := Current_Info (D);
9
10    Turn_Off (D);
11
12    -- The following call raises
13    -- an exception at runtime
14    -- because D is turned off.
15    I := Current_Info (D);
16 end Show_Device;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: cba0100ad5bbb2b6bf00d0847a700271

Runtime output

```
raised CONSTRAINT_ERROR : devices.adb:15 discriminant check failed
```

In this example, by using the variant part, we're preventing information retrieved by an inappropriate call to the `Current_Info` function from being used elsewhere in the application. In fact, if the device is turned off, a call to `Current_Info` raises the `Constraint_Error` exception because the `Info` component isn't accessible. We see that effect in the `Show_Device` procedure: the call to `Current_Info` *fails* (by raising an exception) when the device has just been turned off.

To avoid exceptions at runtime, we must check the device's state before calling `Current_Info`:

Listing 688: show_device.adb

```

1  with Devices; use Devices;
2
3  procedure Show_Device is
4      D : Device;
5      I : Device_Info;
6  begin
7      Turn_On (D);
8
9      if D.State = On then
10         I := Current_Info (D);
11     end if;
12
13     Turn_Off (D);
14
15     if D.State = On then
16         I := Current_Info (D);
17     end if;
18 end Show_Device;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Device
MD5: 62230848af720b156f22c96d59f772d2

Now, no exception is raised, as we only retrieve information from the device when it is turned on — that is, we only call the `Current_Info` function when the `State` discriminant of the object is set to `On`.

Extending record types

We can use variant parts as a means to extend record types. This can be viewed as a static approach to implement type extension — similar to type extension via tagged types, but with clear differences.

Let's say we have a sensor, and we implement a package called `Sensors` that interfaces with that sensor:

Listing 689: sensors.ads

```

1  package Sensors is
2
3      type Sensor is private;
4
5      type Sensor_Info is
6      record
7          Info_1 : Float := 0.0;
8      end record;
9
10     function Current_Info (S : Sensor)
11         return Sensor_Info;
```

(continues on next page)

(continued from previous page)

```
12
13     procedure Display (SI : Sensor_Info);
14
15 private
16
17     type Sensor is null record;
18
19 end Sensors;
```

Listing 690: sensors.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Sensors is
4
5     function Current_Info (S : Sensor)
6         return Sensor_Info is
7         ((Info_1 => 4.0));
8         --      ^^^^
9         --  NOTE: we're returning dummy
10        --      information!
11
12     procedure Display (SI : Sensor_Info) is
13     begin
14         Put_Line ("Info_1 : "
15                 & SI.Info_1'Image);
16     end Display;
17
18 end Sensors;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 140a0d9cbca023de875417409c3f67d9

The Sensor type from the Sensors package has two subprograms: the Current_Info function and the Display procedure. We use those subprograms in the Show_Sensors procedure below:

Listing 691: show_sensors.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Sensors;      use Sensors;
4
5 procedure Show_Sensors is
6     S1 : Sensor;
7 begin
8     Display (Current_Info (S1));
9 end Show_Sensors;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 93aa76da463fea9b4483ed97fa8bcf64

Runtime output

```
Info_1 : 4.00000E+00
```

Now, let's assume that a new model of this sensor is available, and it has additional fea-

tures — e.g., it provides additional information to the user. If we wanted to update the application to be able to handle this new model of the sensor without removing support for the original model, we could convert the `Sensor_Info` type to a tagged type and derive a `Sensor_Info_V2` type from it. (We would probably have to implement a `Sensor_V2` type derived from the `Sensor` type as well.)

Alternatively, we could add a variant part to the `Sensor_Info` type to store the additional information. For example:

Listing 692: sensors.ads

```

1 package Sensors is
2
3   type Sensor_Model is (Sensor_V1,
4                         Sensor_V2);
5
6   type Sensor
7     (Model : Sensor_Model := Sensor_V1) is
8     private;
9
10  type Sensor_Info
11    (Model : Sensor_Model := Sensor_V1) is
12    record
13      Info_1 : Float := 0.0;
14      case Model is
15        when Sensor_V1 =>
16          null;
17        when Sensor_V2 =>
18          Info_2 : Float := 0.0;
19      end case;
20    end record;
21
22  function Current_Info (S : Sensor)
23    return Sensor_Info;
24
25  procedure Display (SI : Sensor_Info);
26
27 private
28
29  type Sensor
30    (Model : Sensor_Model := Sensor_V1) is
31    null record;
32
33 end Sensors;
```

Listing 693: sensors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Sensors is
4
5   function Current_Info (S : Sensor)
6     return Sensor_Info is
7   begin
8     -- Using dummy info for the information
9     -- returned by the function
10    case S.Model is
11      when Sensor_V1 =>
12        return ((Model => Sensor_V1,
13                Info_1 => 4.0));
14      when Sensor_V2 =>
15        return ((Model => Sensor_V2,
```

(continues on next page)

(continued from previous page)

```

16         Info_1 => 8.0,
17         Info_2 => 6.0));
18     end case;
19 end Current_Info;
20
21 procedure Display (SI : Sensor_Info) is
22 begin
23     Put_Line ("Model  : "
24             & SI.Model'Image);
25     Put_Line ("Info_1 : "
26             & SI.Info_1'Image);
27     if SI.Model = Sensor_V2 then
28         Put_Line ("Info_2 : "
29                 & SI.Info_2'Image);
30     end if;
31 end Display;
32
33 end Sensors;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 74198e928e3dfa3a7a7f2786971da8a7

In this new version of the Sensors package, the Model discriminant was added to the Sensor_Info type. If the model is set to version 2 for a specific sensor (i.e., Model = Sensor_V2), a new component (Info_2) is available.

The Current_Info and Display subprograms have been adapted to take this new model into account. In the Current_Info function, we return information for the newer model of the sensor. In the Display procedure, we display the additional information provided by the newer model.

Note that the original test application that makes use of the sensor (Show_Sensors) doesn't require any adaptation:

Listing 694: show_sensors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Sensors;      use Sensors;
4
5 procedure Show_Sensors is
6     S1 : Sensor;
7 begin
8     Display (Current_Info (S1));
9 end Show_Sensors;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 93aa76da463fea9b4483ed97fa8bcf64

Runtime output

```
Info_1 : 4.00000E+00
```

Because we have a default value for the discriminant of the Sensor type, we're essentially making the type *backwards-compatible*, so that users of this type don't have to adapt their code after the update to the Sensors package. Of course, we don't have *binary backwards-compatibility* because the size of the type (Sensor_Info'Size) increases.

Of course, in our test application, we can also use the new model of that sensor:

Listing 695: show_sensors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Sensors;      use Sensors;
4
5 procedure Show_Sensors is
6   S1 : Sensor;
7   S2 : Sensor (Sensor_V2);
8 begin
9   Display (Current_Info (S1));
10  Display (Current_Info (S2));
11 end Show_Sensors;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Variant_Parts.Sensors
MD5: 347d272cddbacf7bf2987aa23014ff0b

Runtime output

```

Model   : SENSOR_V1
Info_1  : 4.00000E+00
Model   : SENSOR_V2
Info_1  : 8.00000E+00
Info_2  : 6.00000E+00
```

In the updated version of the Show_Sensors procedure, we're now using both old and new versions of the sensor.

25.4.9 Per-Object Expressions

In record type declarations, we might want to define a component that makes use of a *name* (page 279) that refers to a *discriminant* (page 458) of the record type, or to the record type itself. An expression where we use such a name is called a per-object expression.

The term "per-object" comes from the fact that, in the component definition, we're referring to a piece of information that will be known just when creating an object of that type. For example, if the per-object expression refers to a discriminant of a type T, the actual value of that discriminant will only be specified when we declare an object of type T. Therefore, the component definition is specific for that individual object — but not necessarily for other objects of the same type, as we might use different values for the discriminant.

The constraint that contains a per-object expression is called a per-object constraint. The actual constraint of that component isn't completely known when we declare the record type, but only later on when an object of that type is created. (Note that the syntax of a constraint includes the parentheses or the keyword **range**.)

In addition to referring to discriminants, per-object expressions can also refer to the record type itself, as we'll see later.

Let's start with a simple record declaration:

Listing 696: rec_per_object_expressions.ads

```

1 package Rec_Per_Object_Expressions is
2
3   type Stack (S : Positive) is private;
4
5 private
```

(continues on next page)

(continued from previous page)

```

6
7  type Integer_Array is
8      array (Positive range <>) of Integer;
9
10 type Stack (S : Positive) is record
11     Arr : Integer_Array (1 .. S);
12     --
13     --
14     --
15     --
16     --      Per-object expression
17     --
18     --      (1 .. S)
19     --      ^^^^^^^
20     --      Per-object constraint
21
22     Top : Natural := 0;
23 end record;
24
25 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression
MD5: e4012454ea886fd429d82159b8d344b7

In this example, we see the Stack record type with a discriminant S. In the declaration of the Arr component of the that type, S is a per-object expression, as it refers to the S discriminant. Also, (1 .. S) is a per-object constraint.

Let's look at another example using *anonymous access types* (page 997):

Listing 697: rec_per_object_expressions.ads

```

1  package Rec_Per_Object_Expressions is
2
3      type T is private;
4
5      type T_Processor (Selected_T : access T) is
6          private;
7
8  private
9
10     type T is null record;
11
12     type T_Container (Selected_T : access T) is
13         null record;
14
15     type T_Processor (Selected_T : access T) is
16         record
17             E : T_Container (Selected_T);
18             --
19             --      Selected_T
20             --      ^^^^^^^^^
21             --      Per-object expression
22             --
23             --      (Selected_T)
24             --      ^^^^^^^^^
25             --      Per-object constraint
26         end record;

```

(continues on next page)

(continued from previous page)

```

27
28 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Access_Discriminant
 MD5: 8b404688be1e103773c28a6977785836

Let's focus on the `T_Processor` type from this example. The `Selected_T` discriminant is being used in the definition of the `E` component. The per-object constraint is `(Selected_T)`. Finally, per-object expressions can also refer to the record type we're declaring. For example:

Listing 698: `rec_per_object_expressions.ads`

```

1 package Rec_Per_Object_Expressions is
2
3   type T is limited private;
4
5 private
6
7   type T_Processor (Selected_T : access T) is
8     null record;
9
10  type T is limited record
11    E : T_Processor (T'Access);
12    --
13    --           T'Access
14    --         ~~~~~~
15    --   Per-object expression
16    --
17    --           (T'Access)
18    --         ~~~~~~
19    --   Per-object constraint
20  end record;
21
22 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Access_Discriminant
 MD5: a67b3034008fdf2a8c5fd1b6da769128

In this example, when we write `T'Access` within the declaration of the `T` record type, the actual value for the `Access` attribute will be known when an object of `T` type is created. In that sense, `T'Access` is a per-object expression — `(T'Access)` is the corresponding per-object constraint.

Note that `T'Access` is referring to the type within a type definition. This is generally treated as a reference to the object being created, the so-called *current instance*.

** In the Ada Reference Manual**

- 3.8 Record Types¹¹⁰

¹¹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

Default value

We can also use per-object expressions to calculate the default value of a record component:

Listing 699: rec_per_object_expressions.ads

```
1 package Rec_Per_Object_Expressions is
2
3   type T (D : Positive) is private;
4
5 private
6
7   type T (D : Positive) is record
8     V : Natural := D - 1;
9     --      ^^^^^
10    --      Per-object expression
11
12     S : Natural := D'Size;
13     --      ^^^^^
14    --      Per-object expression
15   end record;
16
17 end Rec_Per_Object_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Default_Value
MD5: 70454b0b116094a02b897d8d1d0080fb

Here, we calculate the default value of V using the per-object expression `D - 1`, and the default of value of S using the per-object `D'Size`.

The default expression for a component of a discriminated record can be an arbitrary per-object expression. (This contrasts with *important restrictions* (page 515) that exist for per-object constraints, as we discuss later on.) Such expressions might include function calls or uses of any defined operator. For this reason, the following code example is accepted by the compiler:

Listing 700: rec_per_object_expressions.ads

```
1 package Rec_Per_Object_Expressions is
2
3   type Stack (S : Positive) is private;
4
5 private
6
7   type Integer_Array is
8     array (Positive range <>) of Integer;
9
10  type Stack (S : Positive) is record
11    Arr : Integer_Array (1 .. S);
12
13    Top : Natural := 0;
14
15    Overflow_Warning : Positive
16      := S * 9 / 10;
17    --      ^^^^^^^^^
18    --      Per-object expression
19    --      using computation for
20    --      the default expression.
21  end record
22  with
```

(continues on next page)

(continued from previous page)

```

23     Dynamic_Predicate =>
24         Overflow_Warning in
25             (S + 1) / 2 .. S - 1;
26     --
27     -- (S + 1) / 2
28     -- ^^^^^^^^^
29     -- Per-object expression
30     -- using computation.
31     --
32     -- S - 1
33     -- ^^^^^
34     -- Per-object expression
35     -- using computation.
36
37 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Computation
MD5: 6783568fd3e76a85ca7c1cc65ba023c5

In this example, we can identify multiple per-object expressions that use a computation: $S * 9 / 10$, $(S + 1) / 2$, and $S - 1$.

Restrictions

There are some important restrictions on per-object constraints:

1. Per-object range constraints such as $1 \dots T'.Size$ are not allowed.
 - For example, the following code example doesn't compile:

Listing 701: rec_per_object_expressions.ads

```

1  package Rec_Per_Object_Expressions is
2
3      type Bit_Field is
4          array (Positive range <>) of Boolean
5              with Pack;
6
7      type T is record
8          Arr : Bit_Field (1 .. T'.Size);
9              -- ^^^^^
10             -- ERROR: per-object range constraint
11             -- using the Size attribute
12             -- is illegal.
13      end record;
14
15 end Rec_Per_Object_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Range_Constraint
MD5: c2ac9588c1d1adac8c584a0e36a81342

Build output

```

rec_per_object_expressions.ads:8:30: error: in a constraint the
current instance can only be used with an access attribute
gprbuild: *** compilation phase failed

```

2. Within a per-object index constraint or discriminant constraint, each per-object expression must be the name of a discriminant directly, without any further computation.
 - Therefore, we're allowed to write `(1 .. S)` — as we've seen in a previous example —. However, writing `(1 .. S - 1)` would be illegal.
 - For example, the following adaptation to the previous code example doesn't compile:

Listing 702: `rec_per_object_expressions.ads`

```

1 package Rec_Per_Object_Expressions is
2
3     type Stack (S : Positive) is private;
4
5 private
6
7     type Integer_Array is
8         array (Natural range <>) of Integer;
9
10    type Stack (S : Positive) is record
11        Arr : Integer_Array (0 .. S - 1);
12        --      ^^^^^
13        --  ERROR: computation in per-object
14        --      expression is illegal.
15
16        Top : Integer := -1;
17    end record;
18
19 end Rec_Per_Object_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Records.Per_Object_Expressions.Per_Object_Expression_Range_Computation
 MD5: 1224bb63f7953743d84a258226c35c50

Build output

```

rec_per_object_expressions.ads:11:33: error: discriminant in
↳ constraint must appear alone
gprbuild: *** compilation phase failed
```

In this example, using the computation `S - 1` to specify the range of `Arr` isn't permitted. (Note that, *as we've seen before* (page 514), this restriction doesn't apply when the computation is used in a per-object expression that calculates the default value of a component.)

3. We can only use access attributes (`T'Access` and `T'Unchecked_Access`) in per-object constraints.

25.5 Aggregates

25.5.1 Container Aggregates

Note

This feature was introduced in Ada 2022.

A container aggregate is a list of elements — such as `[1, 2, 3]` — that we use to initialize or assign to a container. For example:

Listing 703: show_container_aggregate.adb

```

1 with Ada.Containers.Vectors;
2
3 procedure Show_Container_Aggregate is
4
5     package Float_Vec is new
6         Ada.Containers.Vectors (Positive, Float);
7
8     V : constant Float_Vec.Vector :=
9         [1.0, 2.0, 3.0];
10
11     pragma Unreferenced (V);
12 begin
13     null;
14 end Show_Container_Aggregate;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_Container_Aggregate
MD5: b54cd5800179d4016bbce5a9b10734f2

In this example, `[1.0, 2.0, 3.0]` is a container aggregate that we use to initialize a vector `V`.

We can specify container aggregates in three forms:

- as a null container aggregate, which indicates a container without any elements and is represented by the `[]` syntax;
- as a positional container aggregate, where the elements are simply listed in a sequence (such as `[1, 2]`);
- as a named container aggregate, where a key is indicated for each element of the list (such as `[1 => 10, 2 => 15]`).

Let's look at a complete example:

Listing 704: show_container_aggregate.adb

```

1 with Ada.Containers.Vectors;
2
3 procedure Show_Container_Aggregate is
4
5     package Float_Vec is new
6         Ada.Containers.Vectors (Positive, Float);
7
8     -- Null container aggregate
9     Null_V : constant Float_Vec.Vector :=
10         [];
11
12     -- Positional container aggregate
13     Pos_V : constant Float_Vec.Vector :=
14         [1.0, 2.0, 3.0];
15
16     -- Named container aggregate
17     Named_V : constant Float_Vec.Vector :=
18         [1 => 1.0,
19          2 => 2.0,
20          3 => 3.0];
```

(continues on next page)

(continued from previous page)

```

21
22     pragma Unreferenced (Null_V, Pos_V, Named_V);
23 begin
24     null;
25 end Show_Container_Aggregate;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Simple_
↳ Container_Aggregate
MD5: f00b21da1722669ae92bd5fe4a9a3966

```

In this example, we see the three forms of container aggregates. The difference between positional and named container aggregates is that:

- for positional container aggregates, the vector index is implied by its position;

while

- for named container aggregates, the index (or key) of each element is explicitly indicated.

Also, the named container aggregate in this example (Named_V) is using an index as the name (i.e. it's an indexed aggregate). Another option is to use non-indexed aggregates, where we use actual keys — as we do in maps. For example:

Listing 705: show_named_container_aggregate.adb

```

1  with Ada.Containers.Vectors;
2  with Ada.Containers.Indefinite_Hashed_Maps;
3  with Ada.Strings.Hash;
4
5  procedure Show_Named_Container_Aggregate is
6
7      package Float_Vec is new
8          Ada.Containers.Vectors (Positive, Float);
9
10     package Float_Hashed_Maps is new
11         Ada.Containers.Indefinite_Hashed_Maps
12             (Key_Type      => String,
13              Element_Type  => Float,
14              Hash          => Ada.Strings.Hash,
15              Equivalent_Keys => "=");
16
17     -- Named container aggregate
18     -- using an index
19     Indexed_Named_V : constant Float_Vec.Vector :=
20         [1 => 1.0,
21          2 => 2.0,
22          3 => 3.0];
23
24     -- Named container aggregate
25     -- using a key
26     Keyed_Named_V : constant
27         Float_Hashed_Maps.Map :=
28         ["Key_1" => 1.0,
29          "Key_2" => 2.0,
30          "Key_3" => 3.0];
31
32     pragma Unreferenced (Indexed_Named_V,
33                          Keyed_Named_V);
34 begin

```

(continues on next page)

(continued from previous page)

```

35     null;
36 end Show_Named_Container_Aggregate;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Container_Aggregates.Named_Container_Aggregate
 MD5: 9d117543135e75e66801628ca29e32ef

In this example, `Indexed_Named_V` and `Keyed_Named_V` are both initialized with a named container aggregate. However:

- the container aggregate for `Indexed_Named_V` is an indexed aggregate, so we use an index for each element;

while

- the container aggregate for `Keyed_Named_V` has a key for each element.

Later on, we'll talk about the Aggregate aspect, which allows for defining custom container aggregates for any record type.

i In the Ada Reference Manual

- 4.3.5 Container Aggregates¹¹¹

25.5.2 Record aggregates

We've already seen record aggregates in the *Introduction to Ada* (page 66) course, so this is just a brief overview on the topic.

As we already know, record aggregates can have positional and named component associations. For example, consider this package:

Listing 706: points.ads

```

1 package Points is
2
3     type Point_3D is record
4         X, Y, Z : Integer;
5     end record;
6
7     procedure Display (P : Point_3D);
8
9 end Points;

```

Listing 707: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put_Line ("X => "
8                 & Integer'Image (P.X)
9                 & ",");
10        Put_Line (" Y => "

```

(continues on next page)

¹¹¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-5.html>

(continued from previous page)

```
11         & Integer'Image (P.Y)
12         & ",");
13     Put_Line (" Z => "
14         & Integer'Image (P.Z)
15         & ")");
16 end Display;
17
18 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↳ Rec_Aggregates
MD5: fd01961cf1da9b48d2a6150da30f7377

We can use positional or named record aggregates when assigning to an object P of Point_3D type:

Listing 708: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4     P : Point_3D;
5 begin
6     -- Positional component association
7     P := (0, 1, 2);
8
9     Display (P);
10
11     -- Named component association
12     P := (X => 3,
13         Y => 4,
14         Z => 5);
15
16     Display (P);
17 end Show_Record_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_
↳ Rec_Aggregates
MD5: fc4cff950e31a633ab4e2ae3d21ddc7b

Runtime output

```
(X => 0,
Y => 1,
Z => 2)
(X => 3,
Y => 4,
Z => 5)
```

Also, we can have a mixture of both:

Listing 709: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4     P : Point_3D;
```

(continues on next page)

(continued from previous page)

```

5 begin
6   -- Positional and named component associations
7   P := (3, 4,
8         Z => 5);
9
10  Display (P);
11 end Show_Record_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
 MD5: 493a2a87b4b28dfb0882ad73acf84710

Runtime output

```

(X => 3,
Y => 4,
Z => 5)

```

In this case, only the Z component has a named association, while the other components have a positional association.

Note that a positional association cannot follow a named association, so we cannot write `P := (3, Y => 4, 5);`, for example. Once we start using a named association for a component, we have to continue using it for the remaining components.

In addition, we can choose multiple components at once and assign the same value to them. For that, we use the `|` syntax:

Listing 710: show_record_aggregates.adb

```

1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   -- Multiple component selection
7   P := (X | Y => 5,
8         Z   => 6);
9
10  Display (P);
11 end Show_Record_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
 MD5: a4fde562fb60d290caf46d86b13e694b

Runtime output

```

(X => 5,
Y => 5,
Z => 6)

```

Here, we assign 5 to both X and Y.

** In the Ada Reference Manual**

- 4.3.1 Record Aggregates¹¹²



We can use the `<>` syntax to tell the compiler to use the default value for specific components. However, if there's no default value for specific components, that component isn't initialized to a known value. For example:

Listing 711: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D;
5 begin
6   P := (0, 1, 2);
7   Display (P);
8
9   -- Specifying X component.
10  P := (X => 42,
11        Y => <>,
12        Z => <>);
13  Display (P);
14
15  -- Specifying Y and Z components.
16  P := (X => <>,
17        Y => 10,
18        Z => 20);
19  Display (P);
20 end Show_Record_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
MD5: 25145e7cba5a566c518ac4218e550899

Runtime output

```
(X => 0,
Y => 1,
Z => 2)
(X => 42,
Y => 1,
Z => 2)
(X => 42,
Y => 10,
Z => 20)
```

Here, as the components of `Point_3D` don't have a default value, those components that have `<>` are not initialized:

- when we write `(X => 42, Y => <>, Z => <>)`, only `X` is initialized;
- when we write `(X => <>, Y => 10, Z => 20)` instead, only `X` is uninitialized.

For further reading...

As we've just seen, all components that get a `<>` are uninitialized because the components of `Point_3D` don't have a default value. As no initialization is taking place for

¹¹² <http://www.ada-auth.org/standards/22rm/html/RM-4-3-1.html>

those components of the aggregate, the actual value that is assigned to the record is undefined. In other words, the resulting behavior might depend on the compiler's implementation.

When using GNAT, writing (X => 42, Y => <>, Z => <>) keeps the value of Y and Z intact, while (X => <>, Y => 10, Z => 20) keeps the value of X intact.

If the components of Point_3D had default values, those would have been used. For example, we may change the type declaration of Point_3D and use default values for each component:

Listing 712: points.ads

```

1 package Points is
2
3   type Point_3D is record
4     X : Integer := 10;
5     Y : Integer := 20;
6     Z : Integer := 30;
7   end record;
8
9   procedure Display (P : Point_3D);
10
11 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
MD5: 8a716db129e6f231c4003b77d8b61ea3

Then, writing <> makes use of those default values we've just specified:

Listing 713: show_record_aggregates.adb

```

1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D := (0, 0, 0);
5 begin
6   -- Using default value for
7   -- all components
8   P := (X => <>,
9         Y => <>,
10        Z => <>);
11   Display (P);
12 end Show_Record_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
MD5: e64c6fe4e4b3dbaa084d9b97b4fb971f

Runtime output

```

(X => 10,
Y => 20,
Z => 30)
```

Now, as expected, the default values of each component (10, 20 and 30) are used when we write <>.

Similarly, we can specify a default value for the type of each component. For example, let's declare a `Point_Value` type with a default value — using the `Default_Value` aspect — and use it in the `Point_3D` record type:

Listing 714: points.ads

```
1 package Points is
2
3   type Point_Value is new Float
4     with Default_Value => 99.9;
5
6   type Point_3D is record
7     X : Point_Value;
8     Y : Point_Value;
9     Z : Point_Value;
10  end record;
11
12  procedure Display (P : Point_3D);
13
14 end Points;
```

Listing 715: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5   procedure Display (P : Point_3D) is
6   begin
7     Put_Line ("X => "
8       & Point_Value'Image (P.X)
9       & ",");
10    Put_Line (" Y => "
11      & Point_Value'Image (P.Y)
12      & ",");
13    Put_Line (" Z => "
14      & Point_Value'Image (P.Z)
15      & ")");
16  end Display;
17
18 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
Aggregate_Default_Value
MD5: 508d7f5e7d02da1677485f7d588847f6

Then, writing `<>` makes use of the default value of the `Point_Value` type:

Listing 716: show_record_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Record_Aggregates is
4   P : Point_3D := (0.0, 0.0, 0.0);
5 begin
6   -- Using default value of Point_Value
7   -- for all components
8   P := (X => <>,
9     Y => <>,
10    Z => <>);
```

(continues on next page)

(continued from previous page)

```

11     Display (P);
12 end Show_Record_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_Aggregate_Default_Value
 MD5: 895799077af4a295c250480c32954a2c

Runtime output

```

(X => 9.99000E+01,
 Y => 9.99000E+01,
 Z => 9.99000E+01)

```

In this case, the default value of the `Point_Value` type (99.9) is used for all components when we write `<>`.

others

Also, we can use the **others** selector to assign a value to all components that aren't explicitly mentioned in the aggregate. For example:

Listing 717: show_record_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Record_Aggregates is
4      P : Point_3D;
5  begin
6      -- Specifying X component;
7      -- using 42 for all
8      -- other components.
9      P := (X      => 42,
10           others => 100);
11      Display (P);
12
13      -- Specifying all components
14      P := (others => 256);
15      Display (P);
16 end Show_Record_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Pos_Named_Rec_Aggregates
 MD5: 3146363eb36ab4485c7755794fb78bbc

Runtime output

```

(X => 42,
 Y => 100,
 Z => 100)
(X => 256,
 Y => 256,
 Z => 256)

```

When we write `P := (X => 42, others => 100)`, we're assigning 42 to X and 100 to all other components (Y and Z in this case). Also, when we write `P := (others => 256)`, all components have the same value (256).

Note that writing a specific value in **others** — such as (**others** => 256) — only works when all components have the same type. In this example, all components of `Point_3D` have the same type: **Integer**. If we had components with different types in the components selected by **others**, say **Integer** and **Float**, then (**others** => 256) would trigger a compilation error. For example, consider this package:

Listing 718: `custom_records.ads`

```
1 package Custom_Records is
2
3     type Integer_Float is record
4         A, B : Integer := 0;
5         Y, Z : Float   := 0.0;
6     end record;
7
8 end Custom_Records;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↳ Aggregates_Others
MD5: 875e470aa2cbc5fcfefae649ed5528f6

If we had written an aggregate such as (**others** => 256) for an object of type `Integer_Float`, the value (256) would be OK for components A and B, but not for components Y and Z:

Listing 719: `show_record_aggregates_others.adb`

```
1 with Custom_Records; use Custom_Records;
2
3 procedure Show_Record_Aggregates_Others is
4     Dummy : Integer_Float;
5 begin
6     -- ERROR: components selected by
7     --         others must be of same
8     --         type.
9     Dummy := (others => 256);
10 end Show_Record_Aggregates_Others;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
↳ Aggregates_Others
MD5: d543ee07e24caf63384ab0d140054be2

Build output

```
show_record_aggregates_others.adb:9:14: error: components in "others" choice must
↳ have same type
show_record_aggregates_others.adb:9:24: error: expected type "Standard.Float"
show_record_aggregates_others.adb:9:24: error: found type universal integer
gprbuild: *** compilation phase failed
```

We can fix this compilation error by making sure that **others** only refers to components of the same type:

Listing 720: `show_record_aggregates_others.adb`

```
1 with Custom_Records; use Custom_Records;
2
3 procedure Show_Record_Aggregates_Others is
```

(continues on next page)

(continued from previous page)

```

4   Dummy : Integer_Float;
5   begin
6       -- OK: components selected by
7       --      others have Integer type.
8       Dummy := (Y | Z => 256.0,
9                 others => 256);
10  end Show_Record_Aggregates_Others;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
 ↳Aggregates_Others
 MD5: d01977a49e08d2c6cb6b7788581ed56f

In any case, writing (**others** => <>) is always accepted by the compiler because it simply selects the default value of each component, so the type of those values is unambiguous:

Listing 721: show_record_aggregates_others.adb

```

1   with Custom_Records; use Custom_Records;
2
3   procedure Show_Record_Aggregates_Others is
4       Dummy : Integer_Float;
5   begin
6       Dummy := (others => <>);
7   end Show_Record_Aggregates_Others;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
 ↳Aggregates_Others
 MD5: db9b72ffc933436e76305887276eeafd

This code compiles because <> uses the appropriate default value of each component.

Record discriminants

When a record type has discriminants, they must appear as components of an aggregate of that type. For example, consider this package:

Listing 722: points.ads

```

1   package Points is
2
3       type Point_Dimension is (Dim_1, Dim_2, Dim_3);
4
5       type Point (D : Point_Dimension) is record
6           case D is
7               when Dim_1 =>
8                   X1      : Integer;
9               when Dim_2 =>
10                  X2, Y2   : Integer;
11               when Dim_3 =>
12                  X3, Y3, Z3 : Integer;
13           end case;
14       end record;
15
16       procedure Display (P : Point);
17
18   end Points;

```

Listing 723: points.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Points is
4
5      procedure Display (P : Point) is
6      begin
7          Put_Line (Point_Dimension'Image (P.D));
8
9          case P.D is
10         when Dim_1 =>
11             Put_Line ("  (X => "
12                     & Integer'Image (P.X1)
13                     & ")");
14         when Dim_2 =>
15             Put_Line ("  (X => "
16                     & Integer'Image (P.X2)
17                     & ", "
18                     & " Y => "
19                     & Integer'Image (P.Y2)
20                     & ")");
21         when Dim_3 =>
22             Put_Line ("  (X => "
23                     & Integer'Image (P.X3)
24                     & ", "
25                     & " Y => "
26                     & Integer'Image (P.Y3)
27                     & ", "
28                     & " Z => "
29                     & Integer'Image (P.Z3)
30                     & ")");
31         end case;
32     end Display;
33
34 end Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_
 Aggregate_Discriminant
 MD5: bd71322a65ca50e1eefa0aedd407931a

To write aggregates of the Point type, we have to specify the D discriminant as a component of the aggregate. The discriminant must be included in the aggregate — and must be static — because the compiler must be able to examine the aggregate to determine if it is both complete and consistent. All components must be accounted for one way or another, as usual — but, in addition, references to those components whose existence depends on the discriminant's values must be consistent with the actual discriminant value used in the aggregate. For example, for type Point, an aggregate can only reference the X3, Y3, and Z3 components when Dim_3 is specified for the discriminant D; otherwise, those three components don't exist in that aggregate. Also, the discriminant D must be the first one if we use positional component association. For example:

Listing 724: show_rec_aggregate_discriminant.adb

```

1  with Points; use Points;
2
3  procedure Show_Rec_Aggregate_Discriminant is
4      -- Positional component association
5      P1 : constant Point := (Dim_1, 0);

```

(continues on next page)

(continued from previous page)

```

6
7  -- Named component association
8  P2 : constant Point := (D => Dim_2,
9                          X2 => 3,
10                         Y2 => 4);
11
12 -- Positional / named component association
13 P3 : constant Point := (Dim_3,
14                         X3 => 3,
15                         Y3 => 4,
16                         Z3 => 5);
17 begin
18   Display (P1);
19   Display (P2);
20   Display (P3);
21 end Show_Rec_Aggregate_Discriminant;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Record_Aggregates.Rec_Aggregate_Discriminant
MD5: d487e0c68ea69c3e0f2adb8ac958e31d

Runtime output

```

DIM_1
(X => 0)
DIM_2
(X => 3,
Y => 4)
DIM_3
(X => 3,
Y => 4,
Z => 5)

```

As we see in this example, we can use any component association in the aggregate, as long as we make sure that the discriminants of the type appear as components — and are the first components in the case of positional component association.

25.5.3 Full coverage rules for Aggregates**Note**

This section was originally written by Robert A. Duff and published as [Gem #1: Limited Types in Ada 2005](#)¹¹³.

One interesting feature of Ada are the *full coverage rules* for aggregates. For example, suppose we have a record type:

Listing 725: persons.ads

```

1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 package Persons is
5   type Years is new Natural;

```

(continues on next page)

¹¹³ <https://www.adacore.com/gems/gem-1>

(continued from previous page)

```
6
7  type Person is record
8      Name : Unbounded_String;
9      Age  : Years;
10 end record;
11 end Persons;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪ Full_Coverage_Rules
MD5: 7755bffa8b4473c425ae5075e9c478e9

We can create an object of the type using an aggregate:

Listing 726: show_aggregate_init.adb

```
1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Aggregate_Init is
7
8     X : constant Person :=
9         (Name =>
10            To_Unbounded_String ("John Doe"),
11            Age  => 25);
12 begin
13     null;
14 end Show_Aggregate_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪ Full_Coverage_Rules
MD5: 681e665b76265eff4c4d870ec011ba37

The full coverage rules say that every component of Person must be accounted for in the aggregate. If we later modify type Person by adding a component:

Listing 727: persons.ads

```
1 with Ada.Strings.Unbounded;
2 use  Ada.Strings.Unbounded;
3
4 package Persons is
5     type Years is new Natural;
6
7     type Person is record
8         Name      : Unbounded_String;
9         Age       : Natural;
10        Shoe_Size : Positive;
11    end record;
12 end Persons;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules_Aggregates.
↪ Full_Coverage_Rules
MD5: 5fc5b93748d92932bfc9e0f15c0228b7

and we forget to modify X accordingly, the compiler will remind us. Case statements also have full coverage rules, which serve a similar purpose.

Of course, we can defeat the full coverage rules by using **others** (usually for *array aggregates* (page 531) and case statements, but occasionally useful for *record aggregates* (page 519)):

Listing 728: show_aggregate_init_others.adb

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Aggregate_Init_Others is
7
8     X : constant Person :=
9         (Name =>
10             To_Unbounded_String ("John Doe"),
11             others => 25);
12 begin
13     null;
14 end Show_Aggregate_Init_Others;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Full_Coverage_Rules.Aggregates.
 ↳ Full_Coverage_Rules
 MD5: 6d26de8dd6820682cb9150dcbb40f106

According to the Ada RM, **others** here means precisely the same thing as Age | Shoe_Size. But that's wrong: what **others** really means is "all the other components, including the ones we might add next week or next year". That means you shouldn't use **others** unless you're pretty sure it should apply to all the cases that haven't been invented yet.

Later on, we'll discuss *full coverage rules for limited types* (page 1099).

25.5.4 Array aggregates

We've already discussed array aggregates in the *Introduction to Ada* (page 71) course. Therefore, this section just presents some details about this topic.

In the Ada Reference Manual

- 4.3.3 Array Aggregates¹¹⁴

Positional and named array aggregates

Note

The array aggregate syntax using brackets (e.g.: [1, 2, 3]), which we mention in this section, was introduced in Ada 2022.

Similar to *record aggregates* (page 519), array aggregates can be positional or named. Consider this package:

¹¹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-3.html>

Listing 729: points.ads

```
1 package Points is
2
3     type Point_3D is array (1 .. 3) of Integer;
4
5     procedure Display (P : Point_3D);
6
7 end Points;
```

Listing 730: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put_Line ("X => "
8                 & Integer'Image (P (1))
9                 & ",");
10        Put_Line ("Y => "
11                & Integer'Image (P (2))
12                & ",");
13        Put_Line ("Z => "
14                & Integer'Image (P (3))
15                & ")");
16    end Display;
17
18 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
MD5: d4b3becacc321d20810c3c90f4d8b7ff

We can write positional or named aggregates when assigning to an object P of Point_3D type:

Listing 731: show_array_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Array_Aggregates is
4     P : Point_3D;
5 begin
6     -- Positional component association
7     P := [0, 1, 2];
8
9     Display (P);
10
11    -- Named component association
12    P := [1 => 3,
13         2 => 4,
14         3 => 5];
15
16    Display (P);
17 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 2d65c026639d990e7f6a99f7616d7eb4

Runtime output

```
(X => 0,
 Y => 1,
 Z => 2)
(X => 3,
 Y => 4,
 Z => 5)
```

In this example, we assign a positional array aggregate ([1, 2, 3]) to P. Then, we assign a named array aggregate ([1 => 3, 2 => 4, 3 => 5]) to P. In this case, the *names* are the indices of the components we're assigning to.

We can also assign array aggregates to slices:

Listing 732: show_array_aggregates.adb

```
1 with Points; use Points;
2
3 procedure Show_Array_Aggregates is
4   P : Point_3D := [others => 0];
5 begin
6   -- Positional component association
7   P (2 .. 3) := [1, 2];
8
9   Display (P);
10
11  -- Named component association
12  P (2 .. 3) := [1 => 3,
13               2 => 4];
14
15  Display (P);
16 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: d4e4d3ab4b7d538fc4ef1e92d28e47d9

Runtime output

```
(X => 0,
 Y => 1,
 Z => 2)
(X => 0,
 Y => 3,
 Z => 4)
```

Note that, when using a named array aggregate, the index (*name*) that we use in the aggregate doesn't have to match the slice. In this example, we're assigning the component from index 1 of the aggregate to the component of index 2 of the array P (and so on).

Historically

In the first versions of Ada, we could only write array aggregates using parentheses.

Listing 733: show_array_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4      P : Point_3D;
5  begin
6      -- Positional component association
7      P := (0, 1, 2);
8
9      Display (P);
10
11     -- Named component association
12     P := (1 => 3,
13           2 => 4,
14           3 => 5);
15
16     Display (P);
17 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.
 ↪Array_Aggregates
 MD5: 16df9c01e46623ca735b84167a11a0fd

Runtime output

```

(X => 0,
Y => 1,
Z => 2)
(X => 3,
Y => 4,
Z => 5)

```

This syntax is considered obsolescent since Ada 2022: brackets ([1, 2, 3]) should be used instead.

Null array aggregate**Note**

This feature was introduced in Ada 2022.

We can also write null array aggregates: []. As the name implies, this kind of array aggregate doesn't have any components.

Consider this package:

Listing 734: integer_arrays.ads

```

1  package Integer_Arrays is
2
3      type Integer_Array is
4          array (Positive range <>) of Integer;
5
6      procedure Display (A : Integer_Array);
7
8  end Integer_Arrays;

```

Listing 735: integer_arrays.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Integer_Arrays is
4
5     procedure Display (A : Integer_Array) is
6     begin
7         Put_Line ("Length = "
8                 & A'Length'Image);
9
10        Put_Line "(";
11        for I in A'Range loop
12            Put (" "
13                & I'Image
14                & " => "
15                & A (I)'Image);
16            if I /= A'Last then
17                Put_Line (",");
18            else
19                New_Line;
20            end if;
21        end loop;
22        Put_Line (")");
23    end Display;
24
25 end Integer_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
 ↳ Aggregates_2
 MD5: 8e6e4951c14dcc6e8dea9b6a76064930

We can initialize an object N of Integer_Array type with a null array aggregate:

Listing 736: show_array_aggregates.adb

```

1 with Integer_Arrays; use Integer_Arrays;
2
3 procedure Show_Array_Aggregates is
4     N : constant Integer_Array := [];
5 begin
6     Display (N);
7 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
 ↳ Aggregates_2
 MD5: 188f7b006c08927f8cad83557a5e1cd9

Runtime output

```

Length =  0
(
)

```

In this example, when we call the Display procedure, we confirm that N doesn't have any components.

|, <>, others

We've seen the following syntactic elements when we were discussing *record aggregates* (page 519): `|`, `<>` and **others**. We can apply them to array aggregates as well:

Listing 737: show_array_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4      P : Point_3D;
5  begin
6      -- All components have a value of zero.
7      P := [others => 0];
8
9      Display (P);
10
11     -- Both first and second components have
12     -- a value of three.
13     P := [1 | 2 => 3,
14           3   => 4];
15
16     Display (P);
17
18     -- The default value is used for the first
19     -- component, and all other components
20     -- have a value of five.
21     P := [1      => <>,
22           others => 5];
23
24     Display (P);
25 end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
 ↳ Aggregates
 MD5: 648d68f393107b138c6390c599d3d247

Runtime output

```

(X => 0,
 Y => 0,
 Z => 0)
(X => 3,
 Y => 3,
 Z => 4)
(X => -943676656,
 Y => 5,
 Z => 5)
```

In this example, we use the `|`, `<>` and **others** elements in a very similar way as we did with record aggregates. (See the comments in the code example for more details.)

Note that, as for record aggregates, the `<>` makes use of the default value (if it is available). We discuss this topic in more details *later on* (page 545).

..

We can also use the range syntax `(..)` with array aggregates:

Listing 738: show_array_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4      P : Point_3D;
5  begin
6      -- All components have a value of zero.
7      P := [1 .. 3 => 0];
8
9      Display (P);
10
11     -- Both first and second components have
12     -- a value of three.
13     P := [1 .. 2 => 3,
14           3      => 4];
15
16     Display (P);
17
18     -- The default value is used for the first
19     -- component, and all other components
20     -- have a value of five.
21     P := [1      => <>,
22           2 .. 3 => 5];
23
24     Display (P);
25 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
 MD5: 656f44d37ce676b24e9d512639fd0adc

Runtime output

```

(X => 0,
 Y => 0,
 Z => 0)
(X => 3,
 Y => 3,
 Z => 4)
(X => -177036528,
 Y => 5,
 Z => 5)

```

This example is a variation of the previous one. However, in this case, we're using ranges instead of the `|` and `others` syntax.

Missing components

All aggregate components must have an associated value. If we don't specify a value for a certain component, an exception is raised:

Listing 739: show_array_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4      P : Point_3D;
5  begin

```

(continues on next page)

(continued from previous page)

```
6   P := [1 => 4];
7   -- ERROR: value of components at indices
8   --       2 and 3 are missing
9
10  Display (P);
11  end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates
MD5: 34bbc8e8bd0bd3f8b63d07fa881233bd

Build output

```
show_array_aggregates.adb:6:09: warning: too few elements for type "Point_3D"
↳defined at points.ads:3 [enabled by default]
show_array_aggregates.adb:6:09: warning: expected 3 elements; found 1 element
↳[enabled by default]
show_array_aggregates.adb:6:09: warning: Constraint_Error will be raised at run
↳time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_array_aggregates.adb:6 length check failed
```

We can use **others** to specify a value to all components that haven't been explicitly mentioned in the aggregate:

Listing 740: show_array_aggregates.adb

```
1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4     P : Point_3D;
5  begin
6     P := [1 => 4, others => 0];
7     -- OK: unspecified components have a
8     --     value of zero
9
10    Display (P);
11  end Show_Array_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_
↳Aggregates
MD5: 5f1b7e3778b7d5ec990fba9558495758

Runtime output

```
(X => 4,
Y => 0,
Z => 0)
```

However, **others** can only be used when the range is known — compilation fails otherwise:

Listing 741: show_array_aggregates.adb

```

1 with Integer_Arrays; use Integer_Arrays;
2
3 procedure Show_Array_Aggregates is
4   N1 : Integer_Array := [others => 0];
5   -- ERROR: range is unknown
6
7   N2 : Integer_Array (1 .. 3) := [others => 0];
8   -- OK: range is known
9 begin
10  Display (N1);
11  Display (N2);
12 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates_2
MD5: e185c823ca68e9193a0b12270ffebe61

Build output

```

show_array_aggregates.adb:4:27: error: "others" choice not allowed here
show_array_aggregates.adb:4:27: error: qualify the aggregate with a constrained_
↳ subtype to provide bounds for it
gprbuild: *** compilation phase failed

```

Of course, we could fix the declaration of N1 by specifying a range — e.g. `N1 : Integer_Array (1 .. 10) := [others => 0];`.

Iterated component association**Note**

This feature was introduced in Ada 2022.

We can use an iterated component association to specify an aggregate. This is the general syntax:

```

-- All components have a value of zero
P := [for I in 1 .. 3 => 0];

```

Let's see a complete example:

Listing 742: show_array_aggregates.adb

```

1 with Points; use Points;
2
3 procedure Show_Array_Aggregates is
4   P : Point_3D;
5 begin
6   -- All components have a value of zero
7   P := [for I in 1 .. 3 => 0];
8
9   Display (P);
10
11   -- Both first and second components have
12   -- a value of three

```

(continues on next page)

(continued from previous page)

```

13   P := [for I in 1 .. 3 =>
14         (if I = 1 or I = 2
15          then 3
16          else 4)];
17
18   Display (P);
19
20   -- The first component has a value of 99
21   -- and all other components have a value
22   -- that corresponds to its index
23   P := [1 => 99,
24         for I in 2 .. 3 => I];
25
26   Display (P);
27 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
MD5: 68bddcec76f8431b16d1c090b74c2500

Runtime output

```

(X => 0,
 Y => 0,
 Z => 0)
(X => 3,
 Y => 3,
 Z => 4)
(X => 99,
 Y => 2,
 Z => 3)

```

In this example, we use iterated component associations in different ways:

1. We write a simple iteration ([**for** I **in** 1 .. 3 => 0]).
2. We use a conditional expression in the iteration: [**for** I **in** 1 .. 3 => (**if** I = 1 **or** I = 2 **then** 3 **else** 4)].
3. We use a named association for the first element, and then iterated component association for the remaining components: [1 => 99, **for** I **in** 2 .. 3 => I].

So far, we've used a discrete choice list (in the **for** I **in** Range form) in the iterated component association. We could use an iterator (in the **for** E **of** form) instead. For example:

Listing 743: show_array_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Array_Aggregates is
4    P : Point_3D := [for I in Point_3D'Range => I];
5  begin
6    -- Each component is doubled
7    P := [for E of P => E * 2];
8
9    Display (P);
10
11   -- Each component is increased
12   -- by one

```

(continues on next page)

(continued from previous page)

```

13   P := [for E of P => E + 1];
14
15   Display (P);
16 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates
MD5: 932ebc6e51c2146a726bad68b7f2cad0

Runtime output

```

(X => 2,
 Y => 4,
 Z => 6)
(X => 3,
 Y => 5,
 Z => 7)

```

In this example, we use iterators in different ways:

1. We write `[for E of P => E * 2]` to double the value of each component.
2. We write `[for E of P => E + 1]` to increase the value of each component by one.

Of course, we could write more complex operations on E in the iterators.

Multidimensional array aggregates

So far, we've discussed one-dimensional array aggregates. We can also use the same constructs when dealing with multidimensional arrays. Consider, for example, this package:

Listing 744: matrices.ads

```

1 package Matrices is
2
3   type Matrix is array (Positive range <>,
4                         Positive range <>)
5                       of Integer;
6
7   procedure Display (M : Matrix);
8
9 end Matrices;

```

Listing 745: matrices.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Matrices is
4
5   procedure Display (M : Matrix) is
6
7     procedure Display_Row (M : Matrix;
8                           I : Integer) is
9     begin
10      Put_Line (" (");
11      for J in M'Range (2) loop
12        Put (" "
13           & J'Image
14           & " => "

```

(continues on next page)

(continued from previous page)

```

15         & M (I, J)'Image);
16     if J /= M'Last (2) then
17         Put_Line (",");
18     else
19         New_Line;
20     end if;
21 end loop;
22 Put (" ");
23 end Display_Row;
24
25 begin
26     Put_Line ("Length (1) = "
27             & M'Length (1)'Image);
28     Put_Line ("Length (2) = "
29             & M'Length (2)'Image);
30
31     Put_Line ("(");
32     for I in M'Range (1) loop
33         Display_Row (M, I);
34         if I /= M'Last (1) then
35             Put_Line (",");
36         else
37             New_Line;
38         end if;
39     end loop;
40     Put_Line (")");
41
42 end Display;
43
44 end Matrices;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_
 ↪ Aggregates
 MD5: 55573272f8cc0621eef7c924cfd6366a

We can assign multidimensional aggregates to a matrix M using positional or named component association:

Listing 746: show_array_aggregates.adb

```

1  with Matrices; use Matrices;
2
3  procedure Show_Array_Aggregates is
4      M : Matrix (1 .. 2, 1 .. 3);
5  begin
6      -- Positional component association
7      M := [[0, 1, 2],
8           [3, 4, 5]];
9
10     Display (M);
11
12     -- Named component association
13     M := [[1 => 3,
14           2 => 4,
15           3 => 5],
16          [1 => 6,
17           2 => 7,
18           3 => 8]];
19

```

(continues on next page)

(continued from previous page)

```

20   Display (M);
21
22 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Matrix_Aggregates
MD5: fe3cb6ee62422991b444c32239f72d05

Runtime output

```

Length (1) = 2
Length (2) = 3
(
  (
    1 => 0,
    2 => 1,
    3 => 2
  ),
  (
    1 => 3,
    2 => 4,
    3 => 5
  )
)
Length (1) = 2
Length (2) = 3
(
  (
    1 => 3,
    2 => 4,
    3 => 5
  ),
  (
    1 => 6,
    2 => 7,
    3 => 8
  )
)

```

The first aggregate we use in this example is `[[0, 1, 2], [3, 4, 5]]`. Here, `[0, 1, 2]` and `[3, 4, 5]` are subaggregates of the multidimensional aggregate. Subaggregates don't have a type themselves, but are rather just considered part of a multidimensional aggregate (which, of course, has an array type). In this sense, a subaggregate such as `[0, 1, 2]` is different from a one-dimensional aggregate (such as `[0, 1, 2]`), even though they are written in the same way.

Strings in subaggregates

In the case of matrices using characters, we can use strings in the corresponding array aggregates. Consider this package:

Listing 747: string_lists.ads

```

1 package String_Lists is
2
3   type String_List is array (Positive range <>,
4                             Positive range <>)
5                             of Character;

```

(continues on next page)

(continued from previous page)

```

6
7     procedure Display (SL : String_List);
8
9 end String_Lists;

```

Listing 748: string_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body String_Lists is
4
5     procedure Display (SL : String_List) is
6
7         procedure Display_Row (SL : String_List;
8                                I : Integer) is
9
10            begin
11                Put ("  (");
12                for J in SL'Range (2) loop
13                    Put (SL (I, J));
14                end loop;
15                Put (")");
16            end Display_Row;
17
18            begin
19                Put_Line ("Length (1) = "
20                           & SL'Length (1)'Image);
21                Put_Line ("Length (2) = "
22                           & SL'Length (2)'Image);
23
24                Put_Line ("(");
25                for I in SL'Range (1) loop
26                    Display_Row (SL, I);
27                    if I /= SL'Last (1) then
28                        Put_Line (",");
29                    else
30                        New_Line;
31                    end if;
32                end loop;
33                Put_Line (")");
34            end Display;
35
36 end String_Lists;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_
 ↳ Aggregates
 MD5: aacdbb9aa2f3b6146d8a36ca7581fd18

Then, when assigning to an object SL of String_List type, we can use strings in the aggregates:

Listing 749: show_array_aggregates.adb

```

1 with String_Lists; use String_Lists;
2
3 procedure Show_Array_Aggregates is
4     SL : String_List (1 .. 2, 1 .. 3);
5 begin
6     -- Positional component association
7     SL := ["ABC",

```

(continues on next page)

(continued from previous page)

```

8      "DEF"]);
9
10     Display (SL);
11
12     -- Named component associations
13     SL := [[1 => 'A',
14            2 => 'B',
15            3 => 'C'],
16           [1 => 'D',
17            2 => 'E',
18            3 => 'F']];
19
20     Display (SL);
21
22     SL := [[1 => 'X',
23            2 => 'Y',
24            3 => 'Z'],
25           [others => ' ']];
26
27     Display (SL);
28 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.String_Aggregates
MD5: 82c28e5d8e592403d8909b8eaa1fe356

Runtime output

```

Length (1) = 2
Length (2) = 3
(
  (ABC),
  (DEF)
)
Length (1) = 2
Length (2) = 3
(
  (ABC),
  (DEF)
)
Length (1) = 2
Length (2) = 3
(
  (XYZ),
  ( )
)

```

In the first assignment to SL, we have the aggregate ["ABC", "DEF"], which uses strings as subaggregates. (Of course, we can use a named aggregate and assign characters to the individual components.)

<> and default values

As we indicated earlier, the <> syntax sets a component to its default value — if such a default value is available. If a default value isn't defined, however, the component will remain uninitialized, so that the behavior is undefined. Let's look at more complex example to illustrate this situation. Consider this package, for example:

Listing 750: points.ads

```
1 package Points is
2
3     subtype Point_Value is Integer;
4
5     type Point_3D is record
6         X, Y, Z : Point_Value;
7     end record;
8
9     procedure Display (P : Point_3D);
10
11     type Point_3D_Array is
12         array (Positive range <>) of Point_3D;
13
14     procedure Display (PA : Point_3D_Array);
15
16 end Points;
```

Listing 751: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put ("      (X => "
8             & Point_Value'Image (P.X)
9             & ",");
10        New_Line;
11        Put ("      Y => "
12            & Point_Value'Image (P.Y)
13            & ",");
14        New_Line;
15        Put ("      Z => "
16            & Point_Value'Image (P.Z)
17            & ")");
18    end Display;
19
20    procedure Display (PA : Point_3D_Array) is
21    begin
22        Put_Line ("(");
23        for I in PA'Range (1) loop
24            Put_Line (" "
25                & Integer'Image (I)
26                & " =>");
27            Display (PA (I));
28            if I /= PA'Last (1) then
29                Put_Line (",");
30            else
31                New_Line;
32            end if;
33        end loop;
34        Put_Line (")");
35    end Display;
36
37 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Rec_Array_Aggregates
 MD5: ffaf3745621a30362c6aadaec2c3cef2

Then, let's use `<>` for the array components:

Listing 752: show_record_aggregates.adb

```

1  with Points; use Points;
2
3  procedure Show_Record_Aggregates is
4    PA : Point_3D_Array (1 .. 2);
5  begin
6    PA := [ (X => 3,
7             Y => 4,
8             Z => 5),
9            (X => 6,
10             Y => 7,
11             Z => 8) ];
12    Display (PA);
13
14    -- Array components are
15    -- uninitialized.
16    PA := [1 => <>,
17           2 => <>];
18    Display (PA);
19  end Show_Record_Aggregates;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Rec_Array_Aggregates
 MD5: 4575fead51e24b1a06faf4581efad112

Runtime output

```

(
  1 =>
    (X => 3,
     Y => 4,
     Z => 5),
  2 =>
    (X => 6,
     Y => 7,
     Z => 8)
)
(
  1 =>
    (X => 1670633464,
     Y => 25578,
     Z => 1670407849),
  2 =>
    (X => 25578,
     Y => 1,
     Z => 0)
)
```

Because the record components (of the `Point_3D` type) don't have default values, they remain uninitialized when we write `[1 => <>, 2 => <>]`. (In fact, you may see *garbage* in the values displayed by the `Display` procedure.)

When a default value is specified, it is used whenever `<>` is specified. For example, we could use a type that has the `Default_Value` aspect in its specification:

Listing 753: integer_arrays.ads

```

1 package Integer_Arrays is
2
3     type Value is new Integer
4       with Default_Value => 99;
5
6     type Integer_Array is
7       array (Positive range <>) of Value;
8
9     procedure Display (A : Integer_Array);
10
11 end Integer_Arrays;

```

Listing 754: show_array_aggregates.adb

```

1 with Integer_Arrays; use Integer_Arrays;
2
3 procedure Show_Array_Aggregates is
4   N : Integer_Array (1 .. 4);
5 begin
6   N := [for I in N'Range => Value (I)];
7   Display (N);
8
9   N := [others => <>];
10  Display (N);
11 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates_2
 MD5: 8007fb4af578397d1f07ad85e09ab354

Runtime output

```

Length = 4
(
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4
)
Length = 4
(
  1 => 99,
  2 => 99,
  3 => 99,
  4 => 99
)

```

When writing an aggregate for the Point_3D type, any component that has <> gets the default value of the Point type (99):

For further reading...

Similarly, we could specify the Default_Component_Value aspect (which we discussed [earlier on](#) (page 340)) in the declaration of the array type:

Listing 755: integer_arrays.ads

```

1 package Integer_Arrays is
2
3   type Value is new Integer;
4
5   type Integer_Array is
6     array (Positive range <>) of Value
7     with Default_Component_Value => 9999;
8
9   procedure Display (A : Integer_Array);
10
11 end Integer_Arrays;

```

Listing 756: show_array_aggregates.adb

```

1 with Integer_Arrays; use Integer_Arrays;
2
3 procedure Show_Array_Aggregates is
4   N : Integer_Array (1 .. 4);
5 begin
6   N := [for I in N'Range => Value (I)];
7   Display (N);
8
9   N := [others => <>];
10  Display (N);
11 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates_2
MD5: 3f535bc5ce7f74ab0f0f48098a82c98a

Runtime output

Length = 4

```

(
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4
)

```

Length = 4

```

(
  1 => 9999,
  2 => 9999,
  3 => 9999,
  4 => 9999
)

```

In this case, when writing <> for a component, the value specified in the Default_Component_Value aspect is used.

Finally, we might want to use both Default_Value (which we discussed [previously](#) (page 339)) and Default_Component_Value aspects at the same time. In this case, the value specified in the Default_Component_Value aspect has higher priority:

Listing 757: integer_arrays.ads

```

1 package Integer_Arrays is
2
3   type Value is new Integer
4     with Default_Value => 99;
5
6   type Integer_Array is
7     array (Positive range <>) of Value

```



```

8      with Default_Component_Value => 9999;
9
10     procedure Display (A : Integer_Array);
11
12 end Integer_Arrays;

```

Listing 758: show_array_aggregates.adb

```

1 with Integer_Arrays; use Integer_Arrays;
2
3 procedure Show_Array_Aggregates is
4     N : Integer_Array (1 .. 4);
5 begin
6     N := [for I in N'Range => Value (I)];
7     Display (N);
8
9     N := [others => <>];
10    Display (N);
11 end Show_Array_Aggregates;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Array_Aggregates.Array_Aggregates_2
MD5: e58618b565874acaa99c5d494c2acaa4

Runtime output

```

Length = 4
(
  1 => 1,
  2 => 2,
  3 => 3,
  4 => 4
)
Length = 4
(
  1 => 9999,
  2 => 9999,
  3 => 9999,
  4 => 9999
)

```

Here, 9999 is used when we specify <> for a component.

25.5.5 Extension Aggregates

Extension aggregates provide a convenient way to express an aggregate for a type that extends — adds components to — some existing type (the "ancestor"). Although mainly a matter of convenience, an extension aggregate is essential when we want to express an aggregate for an extension of a private ancestor type, that is, when we don't have compile-time visibility to the ancestor type's components.

In the Ada Reference Manual

- 4.3.2 Extension Aggregates¹¹⁵

¹¹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-2.html>

Assignments to objects of derived types

Before we discuss extension aggregates in more detail, though, let's start with a simple use-case. Let's say we have:

- an object A of tagged type T1, and
- an object B of tagged type T2, which extends T1.

We can initialize object B by:

- copying the T1 specific information from A to B, and
- initializing the T2 specific components of B.

We can translate the description above to the following code:

```
A : T1;
B : T2;
begin
  T1 (B) := A;

  B.Extended_Component_1 := Some_Value;
  -- [...]
```

Here, we use T1 (B) to select the ancestor view of object B, and we copy all the information from A to this part of B. Then, we initialize the remaining components of B. We'll elaborate on this kind of assignments later on.

Example: Points

To present a more concrete example, let's start with a package that defines one, two and three-dimensional point types:

Listing 759: points.ads

```
1 package Points is
2
3   type Point_1D is tagged record
4     X : Float;
5   end record;
6
7   procedure Display (P : Point_1D);
8
9   type Point_2D is new Point_1D with record
10     Y : Float;
11   end record;
12
13   procedure Display (P : Point_2D);
14
15   type Point_3D is new Point_2D with record
16     Z : Float;
17   end record;
18
19   procedure Display (P : Point_3D);
20
21 end Points;
```

Listing 760: points.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
```

(continues on next page)

(continued from previous page)

```
5  procedure Display (P : Point_1D) is
6  begin
7      Put_Line ("X => " & P.X'Image & "");
8  end Display;
9
10 procedure Display (P : Point_2D) is
11 begin
12     Put_Line ("X => " & P.X'Image
13              & ", Y => " & P.Y'Image & "");
14 end Display;
15
16 procedure Display (P : Point_3D) is
17 begin
18     Put_Line ("X => " & P.X'Image
19              & ", Y => " & P.Y'Image
20              & ", Z => " & P.Z'Image & "");
21 end Display;
22
23 end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 0acc05ae2310ab4ba038dfdb6bae0495

Let's now focus on the Show_Points procedure below, where we initialize a two-dimensional point using a one-dimensional point.

Listing 761: show_points.adb

```
1  with Points; use Points;
2
3  procedure Show_Points is
4      P_1D : Point_1D;
5      P_2D : Point_2D;
6  begin
7      P_1D := (X => 0.5);
8      Display (P_1D);
9
10     Point_1D (P_2D) := P_1D;
11     -- Equivalent to: "P_2D.X := P_1D.X;"
12
13     P_2D.Y := 0.7;
14
15     Display (P_2D);
16 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 68ae6fa8e6f779aeb97085bd75e082

Runtime output

```
(X => 5.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01)
```

In this example, we're initializing P_2D using the information stored in P_1D. By writing Point_1D (P_2D) on the left side of the assignment, we specify that we want to limit our focus on the Point_1D view of the P_2D object. Then, we assign P_1D to the Point_1D

view of the P_2D object. This assignment initializes the X component of the P_2D object. The Point_2D specific components are not changed by this assignment. (In other words, this is equivalent to just writing P_2D.X := P_1D.X, as the Point_1D type only has the X component.) Finally, in the next line, we initialize the Y component with 0.7.

Using extension aggregates

Note that, in the assignment to P_1D, we use a record aggregate. Extension aggregates are similar to record aggregates, but they include the **with** keyword — for example: (Obj1 **with** Y => 0.5). This allows us to assign to an object with information from another object Obj1 of a parent type and, in the same expression, set the value of the Y component of the type extension.

Let's rewrite the previous Show_Points procedure using extension aggregates:

Listing 762: show_points.adb

```

1  with Points; use Points;
2
3  procedure Show_Points is
4      P_1D : Point_1D;
5      P_2D : Point_2D;
6  begin
7      P_1D := (X => 0.5);
8      Display (P_1D);
9
10     P_2D := (P_1D with Y => 0.7);
11     Display (P_2D);
12 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 4d03f6a565126b602d6f21fe5ee6dd27

Runtime output

```
(X => 5.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01)
```

When we write P_2D := (P_1D **with** Y => 0.7), we're initializing P_2D using:

- the information from the P_1D object — of Point_1D type, which is an ancestor of the Point_2D type —, and
- the information from the record component association list for the remaining components of the Point_2D type. (In this case, the only remaining component of the Point_2D type is Y.)

We could also specify the type of the extension aggregate. For example, in the previous assignment to P_2D, we could write Point_2D' (...) to indicate that we expect the Point_2D type for the extension aggregate.

```

-- Explicitly state that the type of the
-- extension aggregate is Point_2D:

P_2D := Point_2D'(P_1D with Y => 0.7);
```

Also, we don't have to use named association in extension aggregates. We could just use positional association instead. Therefore, we could simplify the assignment to P_2D in the previous example by just writing:

```
P_2D := (P_1D with 0.7);
```

More extension aggregates

We can use extension aggregates for descendants of the `Point_2D` type as well. For example, let's extend our previous code example by declaring an object of `Point_3D` type (called `P_3D`) and use extension aggregates in assignments to this object:

Listing 763: `show_points.adb`

```
1 with Points; use Points;
2
3 procedure Show_Points is
4   P_1D : Point_1D;
5   P_2D : Point_2D;
6   P_3D : Point_3D;
7 begin
8   P_1D := (X => 0.5);
9   Display (P_1D);
10
11   P_2D := (P_1D with Y => 0.7);
12   Display (P_2D);
13
14   P_3D := (P_2D with Z => 0.3);
15   Display (P_3D);
16
17   P_3D := (P_1D with Y | Z => 0.1);
18   Display (P_3D);
19 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 2ec6831557c43f697bffce8496962b53

Runtime output

```
(X => 5.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 5.00000E-01, Y => 1.00000E-01, Z => 1.00000E-01)
```

In the first assignment to `P_3D` in the example above, we're initializing this object with information from `P_2D` and specifying the value of the `Z` component. Then, in the next assignment to the `P_3D` object, we're using an aggregate with information from `P_1` and specifying values for the `Y` and `Z` components. (Just as a reminder, we can write `Y | Z => 0.1` to assign 0.1 to both `Y` and `Z` components.)

with others

Other versions of extension aggregates are possible as well. For example, we can combine keywords and write `with` `others` to focus on all remaining components of an extension aggregate.

Listing 764: `show_points.adb`

```
1 with Points; use Points;
2
3 procedure Show_Points is
```

(continues on next page)

(continued from previous page)

```

4   P_1D : Point_1D;
5   P_2D : Point_2D;
6   P_3D : Point_3D;
7   begin
8     P_1D := (X => 0.5);
9     P_2D := (P_1D with Y => 0.7);
10
11    -- Initialize P_3D with P_1D and set other
12    -- components to 0.6.
13    --
14    P_3D := (P_1D with others => 0.6);
15    Display (P_3D);
16
17    -- Initialize P_3D with P_2D, and other
18    -- components with their default value.
19    --
20    P_3D := (P_2D with others => <>);
21    Display (P_3D);
22  end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 0594586fc59ead106258cef8682927e9

Runtime output

```

(X => 5.00000E-01, Y => 6.00000E-01, Z => 6.00000E-01)
(X => 5.00000E-01, Y => 7.00000E-01, Z => 2.15485E-26)

```

In this example, the first assignment to P_3D has an aggregate with information from P_1D, while the remaining components — in this case, Y and Z — are just set to 0.6.

Continuing with this example, in the next assignment to P_3D, we're using information from P_2 in the extension aggregate. This covers the Point_2D part of the P_3D object — components X and Y, to be more specific. The Point_3D specific components of P_3D — component Z in this case — receive their corresponding default value. In this specific case, however, we haven't specified a default value for component Z in the declaration of the Point_3D type, so we cannot rely on any specific value being assigned to that component when using **others** => <>.

with null record

We can also use extension aggregates with null records. Let's focus on the P_3D_Ext object of Point_3D_Ext type. This object is declared in the Show_Points procedure of the next code example.

Listing 765: points-extensions.ads

```

1  package Points.Extensions is
2
3     type Point_3D_Ext is new
4       Point_3D with null record;
5
6  end Points.Extensions;

```

Listing 766: show_points.adb

```

1 with Points;           use Points;
2 with Points.Extensions; use Points.Extensions;
3
4 procedure Show_Points is
5   P_3D      : Point_3D;
6   P_3D_Ext : Point_3D_Ext;
7 begin
8   P_3D := (X => 0.0, Y => 0.5, Z => 0.4);
9
10  P_3D_Ext := (P_3D with null record);
11  Display (P_3D_Ext);
12 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: 8ec3ddb3a1f2a6e550ac4d622e97124c

Runtime output

```
(X => 0.000000E+00, Y => 5.000000E-01, Z => 4.000000E-01)
```

The P_3D_Ext object is of Point_3D_Ext type, which is declared in the Points.Extensions package and derived from the Point_3D type. Note that we're not extending Point_3D_Ext with new components, but using a null record instead in the declaration. Therefore, as the Point_3D_Ext type doesn't own any new components, we just write (P_3D with null record) to initialize the P_3D_Ext object.

Extension aggregates and descendent types

In the examples above, we've been initializing objects of descendent types by using objects of ascending types in extension aggregates. We could, however, do the opposite and initialize objects of ascending types using objects of descendent type in extension aggregates. Consider this code example:

Listing 767: show_points.adb

```

1 with Points; use Points;
2
3 procedure Show_Points is
4   P_2D : Point_2D;
5   P_3D : Point_3D;
6 begin
7   P_3D := (X => 0.5, Y => 0.7, Z => 0.3);
8   Display (P_3D);
9
10  P_2D := (Point_1D (P_3D) with Y => 0.3);
11  Display (P_2D);
12 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Extension_Aggregates.Extension_Aggregate_Points
MD5: ae5e88a36c58b1eb495d5ba8752e50e7

Runtime output

```
(X => 5.00000E-01, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 5.00000E-01, Y => 3.00000E-01)
```

Here, we're using `Point_1D` (`P_3D`) to select the `Point_1D` view of an object of `Point_3D` type. At this point, we have specified the `Point_1D` part of the aggregate, so we still have to specify the remaining components of the `Point_2D` type — the `Y` component, to be more specific. When we do that, we get the appropriate aggregate for the `Point_2D` type. In summary, by carefully selecting the appropriate view, we're able to initialize an object of ascending type (`Point_2D`), which contains less components, using an object of a descendent type (`Point_3D`), which contains more components.

25.5.6 Delta Aggregates

Note

This feature was introduced in Ada 2022.

Previously, we've discussed *extension aggregates* (page 553), which are used to assign an object `Obj_From` of a tagged type to an object `Obj_To` of a descendent type.

We may want also to assign an object `Obj_From` of to an object `Obj_To` of the same type, but change some of the components in this assignment. To do this, we use delta aggregates.

Delta Aggregates for Tagged Records

Let's reuse the `Points` package from a previous example:

Listing 768: `points.ads`

```
1 package Points is
2
3     type Point_1D is tagged record
4         X : Float;
5     end record;
6
7     type Point_2D is new Point_1D with record
8         Y : Float;
9     end record;
10
11    type Point_3D is new Point_2D with record
12        Z : Float;
13    end record;
14
15    procedure Display (P : Point_3D);
16
17 end Points;
```

Listing 769: `points.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put_Line ("(X => " & P.X'Image
8                 & ", Y => " & P.Y'Image
9                 & ", Z => " & P.Z'Image & ")");
```

(continues on next page)

(continued from previous page)

```

10     end Display;
11
12 end Points;

```

Listing 770: show_points.adb

```

1  with Points; use Points;
2
3  procedure Show_Points is
4      P1, P2, P3 : Point_3D;
5  begin
6      P1 := (X => 0.5, Y => 0.7, Z => 0.3);
7      Display (P1);
8
9      P2 := (P1 with delta X => 1.0);
10     Display (P2);
11
12     P3 := (P1 with delta X => 0.2, Y => 0.3);
13     Display (P3);
14 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Tagged
MD5: 23e9f53d626e32fc0524abfa0a437dbf

Runtime output

```

(X => 5.00000E-01, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 1.00000E+00, Y => 7.00000E-01, Z => 3.00000E-01)
(X => 2.00000E-01, Y => 3.00000E-01, Z => 3.00000E-01)

```

Here, we assign P1 to P2, but change the X component. Also, we assign P1 to P3, but change the X and Y components.

We can use class-wide types with delta aggregates. Consider this example:

Listing 771: show_points.adb

```

1  with Points; use Points;
2
3  procedure Show_Points is
4
5      P_3D : Point_3D;
6
7      function Reset (P_2D : Point_2D'Class)
8          return Point_2D'Class is
9          ((P_2D with delta X | Y => 0.0));
10
11  begin
12      P_3D := (X => 0.1, Y => 0.2, Z => 0.3);
13      Display (P_3D);
14
15      P_3D := Point_3D (Reset (P_3D));
16      Display (P_3D);
17
18 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Tagged
 MD5: dca144fe420dd37e224d089458f9e8a8

Runtime output

```
(X => 1.00000E-01, Y => 2.00000E-01, Z => 3.00000E-01)
(X => 0.00000E+00, Y => 0.00000E+00, Z => 3.00000E-01)
```

In this example, the Reset function returns an object of `Point_2D'Class` where all components of `Point_2D'Class` type are zero. We call the Reset function for the `P_3D` object of `Point_3D` type, so that only the Z component remains untouched.

Note that we use the syntax `X | Y` in the body of the Reset function and assign the same value to both components.

For further reading...

We could have implemented Reset as a procedure — in this case, without using delta aggregates:

Listing 772: show_points.adb

```
1 with Points; use Points;
2
3 procedure Show_Points is
4
5     P_3D : Point_3D;
6
7     procedure Reset
8       (P_2D : in out Point_2D'Class) is
9     begin
10       Point_2D (P_2D) := (others => 0.0);
11     end Reset;
12
13 begin
14   P_3D := (X => 0.1, Y => 0.2, Z => 0.3);
15   Display (P_3D);
16
17   Reset (P_3D);
18   Display (P_3D);
19
20 end Show_Points;
```

Delta Aggregates for Non-Tagged Records

The examples above use tagged types. We can also use delta aggregates with non-tagged types. Let's rewrite the Points package and convert `Point_3D` to a non-tagged record type.

Listing 773: points.ads

```
1 package Points is
2
3   type Point_3D is record
4     X : Float;
5     Y : Float;
6     Z : Float;
7   end record;
8
9   procedure Display (P : Point_3D);
```

(continues on next page)

(continued from previous page)

```

10
11 end Points;

```

Listing 774: points.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Points is
4
5     procedure Display (P : Point_3D) is
6     begin
7         Put_Line ("(X => " & P.X'Image
8                 & ", Y => " & P.Y'Image
9                 & ", Z => " & P.Z'Image & ")");
10    end Display;
11
12 end Points;

```

Listing 775: show_points.adb

```

1 with Points; use Points;
2
3 procedure Show_Points is
4     P1, P2, P3 : Point_3D;
5 begin
6     P1 := (X => 0.5, Y => 0.7, Z => 0.3);
7     Display (P1);
8
9     P2 := (P1 with delta X => 1.0);
10    Display (P2);
11
12    P3 := (P1 with delta X => 0.2, Y => 0.3);
13    Display (P3);
14 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Non_Tagged
MD5: 1f12f33ac0a84919978c56d04f479e35

Runtime output

```

(X =>  5.00000E-01, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  1.00000E+00, Y =>  7.00000E-01, Z =>  3.00000E-01)
(X =>  2.00000E-01, Y =>  3.00000E-01, Z =>  3.00000E-01)

```

In this example, Point_3D is a non-tagged type. Note that we haven't changed anything in the Show_Points procedure: it still works as it did with tagged types.

Delta Aggregates for Arrays

We can use delta aggregates for arrays. Let's change the declaration of Point_3D and use an array to represent a 3-dimensional point:

Listing 776: points.ads

```

1 package Points is
2
3     type Float_Array is

```

(continues on next page)

(continued from previous page)

```

4     array (Positive range <>) of Float;
5
6     type Point_3D is new Float_Array (1 .. 3);
7
8     procedure Display (P : Point_3D);
9
10    end Points;

```

Listing 777: points.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Points is
4
5      procedure Display (P : Point_3D) is
6      begin
7          Put ("(");
8          for I in P'Range loop
9              Put (I'Image
10                 & " => "
11                 & P (I)'Image);
12          end loop;
13          Put_Line (")");
14      end Display;
15
16  end Points;

```

Listing 778: show_points.adb

```

1  with Points; use Points;
2
3  procedure Show_Points is
4      P1, P2, P3 : Point_3D;
5  begin
6      P1 := [0.5, 0.7, 0.3];
7      Display (P1);
8
9      P2 := [P1 with delta 1 => 1.0];
10     Display (P2);
11
12     P3 := [P1 with delta 1 => 0.2, 2 => 0.3];
13     -- Alternatively:
14     -- P3 := [P1 with delta 1 .. 2 => 0.2, 0.3];
15
16     Display (P3);
17 end Show_Points;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
 ↳ Aggregates_Array
 MD5: 06293882e5dd020f56fbced6bc03ccf0

Runtime output

```

( 1 => 5.00000E-01 2 => 7.00000E-01 3 => 3.00000E-01)
( 1 => 1.00000E+00 2 => 7.00000E-01 3 => 3.00000E-01)
( 1 => 2.00000E-01 2 => 3.00000E-01 3 => 3.00000E-01)

```

The implementation of Show_Points in this example is very similar to the version where use a record type. In this case, we:

- assign P1 to P2, but change the first component, and
- we assign P1 to P3, but change the first and second components.

Using slices

In the assignment to P3, we can either specify each component of the delta individually or use a slice: both forms are equivalent. Also, we can use slices to assign the same number to multiple components:

Listing 779: show_points.adb

```
1 with Points; use Points;
2
3 procedure Show_Points is
4   P1, P3 : Point_3D;
5 begin
6   P1 := [0.5, 0.7, 0.3];
7   Display (P1);
8
9   P3 := [P1 with delta
10         P3'First + 1 .. P3'Last => 0.0];
11   Display (P3);
12 end Show_Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_
Aggregates_Array
MD5: 0a00e17b2d803f23edc728969d663c59

Runtime output

```
( 1 => 5.00000E-01 2 => 7.00000E-01 3 => 3.00000E-01)
( 1 => 5.00000E-01 2 => 0.00000E+00 3 => 0.00000E+00)
```

In this example, we're assigning P1 to P3, but resetting all components of the array starting by the second one.

Multiple components

We can also assign multiple components or slices:

Listing 780: float_arrays.ads

```
1 package Float_Arrays is
2
3   type Float_Array is
4     array (Positive range <>) of Float;
5
6   procedure Display (P : Float_Array);
7
8 end Float_Arrays;
```

Listing 781: float_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Float_Arrays is
4
5   procedure Display (P : Float_Array) is
```

(continues on next page)

(continued from previous page)

```

6   begin
7
8       Put ("");
9       for I in P'Range loop
10          Put (I'Image
11              & " => "
12              & P (I)'Image);
13      end loop;
14      Put_Line ("");
15
16  end Display;
17
18 end Float_Arrays;

```

Listing 782: show_multiple_delta_slices.adb

```

1  with Float_Arrays; use Float_Arrays;
2
3  procedure Show_Multiple_Delta_Slices is
4
5      P1, P2 : Float_Array (1 .. 5);
6
7  begin
8      P1 := [1.0, 2.0, 3.0, 4.0, 5.0];
9      Display (P1);
10
11      P2 := [P1 with delta
12            P2'First + 1 .. P2'Last - 2 => 0.0,
13            P2'Last - 1 .. P2'Last => 0.2];
14      Display (P2);
15  end Show_Multiple_Delta_Slices;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Aggregates.Delta_Aggregates.Delta_Aggregates_Array
MD5: 37063cd1c6cd46522d8e5b0df7b5741b

Runtime output

```

( 1 => 1.00000E+00 2 => 2.00000E+00 3 => 3.00000E+00 4 => 4.00000E+00 5 => 5.
->00000E+00)
( 1 => 1.00000E+00 2 => 0.00000E+00 3 => 0.00000E+00 4 => 2.00000E-01 5 => 2.
->00000E-01)

```

In this example, we have two arrays P1 and P2 of Float_Array type. We assign P1 to P2, but change:

- the second to the last-but-two components to 0.0, and
- the last-but-one and last components to 0.2.

** In the Ada Reference Manual**

- [Delta Aggregates](#)¹¹⁶

¹¹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-3-4.html>

25.6 Arrays

25.6.1 Array constraints

Array constraints are important in the declaration of an array because they define the total size of the array. In fact, arrays must always be constrained. In this section, we start our discussion with unconstrained array types, and then continue with constrained arrays and arrays types. Finally, we discuss the differences between unconstrained arrays and vectors.

In the Ada Reference Manual

- 3.6 Array Types¹¹⁷

Unconstrained array types

In the *Introduction to Ada course* (page 78), we've seen that we can declare array types whose bounds are not fixed: in that case, the bounds are provided when creating objects of those types. For example:

Listing 783: measurement_defs.ads

```

1 package Measurement_Defs is
2
3     type Measurements is
4       array (Positive range <>) of Float;
5     --      ^ Bounds are of type Positive,
6     --      but not known at this point.
7
8 end Measurement_Defs;
```

Listing 784: show_measurements.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Measurement_Defs; use Measurement_Defs;
4
5 procedure Show_Measurements is
6   M : Measurements (1 .. 10);
7   --      ^ Providing bounds here!
8 begin
9   Put_Line ("First index: " & M'First'Image);
10  Put_Line ("Last index:  " & M'Last'Image);
11 end Show_Measurements;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Unconstrained_
 ↳ Array_Type
 MD5: a5cdc74dd61e36476431cf675452d1d5

Build output

```
show_measurements.adb:6:04: warning: variable "M" is read but never assigned [-  

↳ gnatwv]
```

Runtime output

¹¹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

```
First index: 1
Last index: 10
```

In this example, the `Measurements` array type from the `Measurement_Defs` package is unconstrained. In the `Show_Measurements` procedure, we declare a constrained object (`M`) of this type.

Constrained arrays

The [Introduction to Ada course](#) (page 79) highlights the fact that the bounds are fixed once an object is declared:

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime. This allows Ada to implement unconstrained arrays efficiently; instances can be stored on the stack and do not require heap allocation as in languages like Java.

In the `Show_Measurements` procedure above, once we declare `M`, its bounds are fixed for the whole lifetime of `M`. We cannot *add* another component to this array. In other words, `M` will have 10 components for its whole lifetime:

```
M : Measurements (1 .. 10);
--               ^^^^^^^
--   Bounds cannot be changed!
```

Constrained array types

Note that we could declare constrained array types. Let's rework the previous example:

Listing 785: `measurement_defs.ads`

```
1 package Measurement_Defs is
2
3     type Measurements is
4       array (1 .. 10) of Float;
5     --     ^ Bounds are of known and fixed.
6
7 end Measurement_Defs;
```

Listing 786: `show_measurements.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Measurement_Defs; use Measurement_Defs;
4
5 procedure Show_Measurements is
6   M : Measurements;
7   --     ^ We cannot change the
8   --     bounds here!
9 begin
10   Put_Line ("First index: " & M'First'Image);
11   Put_Line ("Last index:  " & M'Last'Image);
12 end Show_Measurements;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Constrained_
         ↪ Array_Type
MD5: 4741986fdf4dab731baa001b6e60c345
```

Build output


```
show_measurements.adb:6:04: warning: variable "M" is read but never assigned [-  
↳gnatwv]
```

Runtime output

```
First index: 1  
Last index: 10
```

In this case, the bounds of the Measurements type are fixed. Now, we cannot specify the bounds (or change them) in the declaration of the M array, as they have already been defined in the type declaration.

Unconstrained Arrays vs. Vectors

If you need, however, the flexibility of increasing the length of an array, you could use the language-defined Vector type instead. This is how we could rewrite the previous example using vectors:

Listing 787: measurement_defs.ads

```
1 with Ada.Containers; use Ada.Containers;  
2 with Ada.Containers.Vectors;  
3  
4 package Measurement_Defs is  
5  
6     package Vectors is new Ada.Containers.Vectors  
7         (Index_Type => Positive,  
8          Element_Type => Float);  
9  
10    subtype Measurements is Vectors.Vector;  
11  
12 end Measurement_Defs;
```

Listing 788: show_measurements.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;  
2  
3 with Measurement_Defs; use Measurement_Defs;  
4  
5 procedure Show_Measurements is  
6     use Measurement_Defs.Vectors;  
7  
8     M : Measurements := To_Vector (10);  
9     --           ^ Creating 10-element  
10    --           vector.  
11 begin  
12     Put_Line ("First index: "  
13              & M.First_Index'Image);  
14     Put_Line ("Last index: "  
15              & M.Last_Index'Image);  
16  
17     Put_Line ("Adding element...");  
18     M.Append (1.0);  
19  
20     Put_Line ("First index: "  
21              & M.First_Index'Image);  
22     Put_Line ("Last index: "  
23              & M.Last_Index'Image);  
24 end Show_Measurements;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Constraints.Unconstrained_
 ↪ Array_Type_Vs_Vector
 MD5: afec7a4b898392be4dd1f60e1519da88

Runtime output

```
First index: 1
Last index: 10
Adding element...
First index: 1
Last index: 11
```

In the declaration of `M` in this example, we're creating a 10-element vector by calling `To_Vector` and specifying the element count. Later on, with the call to `Append`, we're increasing the length of the `M` to 11 elements.

As you might expect, the flexibility of vectors comes with a price: every time we add an element that doesn't fit in the current capacity of the vector, the container has to reallocate memory in the background due to that new element. Therefore, arrays are more efficient, as the memory allocation only happens once for each object.

In the Ada Reference Manual

- [3.6 Array Types](#)¹¹⁸
- [A.18.2 The Generic Package Containers.Vectors](#)¹¹⁹

25.6.2 Multidimensional Arrays

So far, we've discussed unidimensional arrays, since they are very common in Ada. However, Ada also supports multidimensional arrays using the same facilities as for unidimensional arrays. For example, we can use the `First`, `Last`, **Range** and `Length` attributes for each dimension of a multidimensional array. This section presents more details on this topic.

To create a multidimensional array, we simply separate the ranges of each dimension with a comma. The following example presents the one-dimensional array `A1`, the two-dimensional array `A2` and the three-dimensional array `A3`:

Listing 789: multidimensional_arrays_decl.ads

```
1 package Multidimensional_Arrays_Decl is
2
3   A1 : array (1 .. 10) of Float;
4   A2 : array (1 .. 5, 1 .. 10) of Float;
5   --           ^ first dimension
6   --           ^ second dimension
7   A3 : array (1 .. 2, 1 .. 5, 1 .. 10) of Float;
8   --           ^ first dimension
9   --           ^ second dimension
10  --           ^ third dimension
11 end Multidimensional_Arrays_Decl;
```

Code block metadata

¹¹⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

¹¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↳Multidimensional_Arrays
MD5: 928243b293c67a078d729c3cac68bb92

The two-dimensional array A2 has 5 components in the first dimension and 10 components in the second dimension. The three-dimensional array A3 has 2 components in the first dimension, 5 components in the second dimension, and 10 components in the third dimension. Note that the ranges we've selected for A1, A2 and A3 are completely arbitrary. You may select ranges for each dimension that are the most appropriate in the context of your application. Also, the number of dimensions is not limited to three, so you could declare higher-dimensional arrays if needed.

We can use the Length attribute to retrieve the length of each dimension. We use an integer value in parentheses to specify which dimension we're referring to. For example, if we write A'Length (2), we're referring to the length of the second dimension of a multidimensional array A. Note that A'Length is equivalent to A'Length (1). The same equivalence applies to other array-related attributes such as First, Last and Range.

Let's use the Length attribute for the arrays we declared in the Multidimensional_Arrays_Decl package:

Listing 790: show_multidimensional_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Multidimensional_Arrays_Decl;
4 use Multidimensional_Arrays_Decl;
5
6 procedure Show_Multidimensional_Arrays is
7 begin
8   Put_Line ("A1'Length:      "
9             & A1'Length'Image);
10  Put_Line ("A1'Length (1): "
11            & A1'Length (1)'Image);
12  Put_Line ("A2'Length (1): "
13            & A2'Length (1)'Image);
14  Put_Line ("A2'Length (2): "
15            & A2'Length (2)'Image);
16  Put_Line ("A3'Length (1): "
17            & A3'Length (1)'Image);
18  Put_Line ("A3'Length (2): "
19            & A3'Length (2)'Image);
20  Put_Line ("A3'Length (3): "
21            & A3'Length (3)'Image);
22 end Show_Multidimensional_Arrays;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↳Multidimensional_Arrays
MD5: 70b9b8df7e46302b92613fa484ef71ca

Runtime output

```
A1'Length:      10
A1'Length (1):  10
A2'Length (1):   5
A2'Length (2):  10
A3'Length (1):   2
A3'Length (2):   5
A3'Length (3):  10
```

As this simple example shows, we can easily retrieve the length of each dimension. Also, as we've just mentioned, `A1'Length` is equal to `A1'Length (1)`.

Let's consider an application where we make hourly measurements for the first 12 hours of the day, on each day of the week. We can create a two-dimensional array type called `Measurements` to store this data. Also, we can have three procedures for this array:

- `Show_Indices`, which presents the indices (days and hours) of the two-dimensional array;
- `Show_Values`, which presents the values stored in the array; and
- `Reset`, which resets each value of the array.

This is the complete code for this application:

Listing 791: `measurement_defs.ads`

```

1 package Measurement_Defs is
2
3   type Days is
4     (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
5
6   type Hours is range 0 .. 11;
7
8   subtype Measurement is Float;
9
10  type Measurements is
11    array (Days, Hours) of Measurement;
12
13  procedure Show_Indices (M : Measurements);
14
15  procedure Show_Values (M : Measurements);
16
17  procedure Reset (M : out Measurements);
18
19 end Measurement_Defs;
```

Listing 792: `measurement_defs.adb`

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 package body Measurement_Defs is
4
5   procedure Show_Indices (M : Measurements) is
6   begin
7     Put_Line ("---- Indices ----");
8
9     for D in M'Range (1) loop
10      Put (D'Image & " ");
11
12      for H in M'First (2) ..
13        M'Last (2) - 1
14      loop
15        Put (H'Image & " ");
16      end loop;
17      Put_Line (M'Last (2)'Image);
18    end loop;
19  end Show_Indices;
20
21  procedure Show_Values (M : Measurements) is
22  package H_IO is
23    new Ada.Text_IO.Integer_IO (Hours);
24  package M_IO is
```

(continues on next page)

(continued from previous page)

```

25     new Ada.Text_IO.Float_IO (Measurement);
26
27     procedure Set_IO_Defaults is
28     begin
29         H_IO.Default_Width := 5;
30
31         M_IO.Default_Fore  := 1;
32         M_IO.Default_Aft   := 2;
33         M_IO.Default_Exp   := 0;
34     end Set_IO_Defaults;
35 begin
36     Set_IO_Defaults;
37
38     Put_Line ("---- Values ----");
39     Put (" ");
40     for H in M'Range (2) loop
41         H_IO.Put (H);
42     end loop;
43     New_Line;
44
45     for D in M'Range (1) loop
46         Put (D'Image & " ");
47
48         for H in M'Range (2) loop
49             M_IO.Put (M (D, H));
50             Put (" ");
51         end loop;
52         New_Line;
53     end loop;
54 end Show_Values;
55
56 procedure Reset (M : out Measurements) is
57 begin
58     M := (others => (others => 0.0));
59 end Reset;
60
61 end Measurement_Defs;

```

Listing 793: show_measurements.adb

```

1  with Measurement_Defs; use Measurement_Defs;
2
3  procedure Show_Measurements is
4      M : Measurements;
5  begin
6      Reset (M);
7      Show_Indices (M);
8      Show_Values (M);
9  end Show_Measurements;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
 ↳Multidimensional_Measurements
 MD5: bcffa3913007bd9152149ad9616842b8

Runtime output

```

---- Indices ----
MON 0 1 2 3 4 5 6 7 8 9 10 11
TUE 0 1 2 3 4 5 6 7 8 9 10 11

```

(continues on next page)

(continued from previous page)

```

WED 0 1 2 3 4 5 6 7 8 9 10 11
THU 0 1 2 3 4 5 6 7 8 9 10 11
FRI 0 1 2 3 4 5 6 7 8 9 10 11
SAT 0 1 2 3 4 5 6 7 8 9 10 11
SUN 0 1 2 3 4 5 6 7 8 9 10 11
---- Values ----
      0      1      2      3      4      5      6      7      8      9      10      11
MON 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
TUE 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
WED 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
THU 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
FRI 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SAT 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SUN 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00

```

We recommend that you spend some time analyzing this example. Also, we'd like to highlight the following aspects:

- We access a value from a multidimensional array by using commas to separate the index values within the parentheses. For example: `M (D, H)` allows us to access the value on day `D` and hour `H` from the multidimensional array `M`.
- To loop over the multidimensional array `M`, we write `for D in M'Range (1) loop` and `for H in M'Range (2) loop` for the first and second dimensions, respectively.
- To reset all values of the multidimensional array, we use an aggregate with this form: `(others => (others => 0.0))`.

In the Ada Reference Manual

- [3.6 Array Types¹²⁰](#)
- [3.6.2 Operations of Array Types¹²¹](#)

Unconstrained Multidimensional Arrays

Previously, we've discussed unconstrained arrays for the unidimensional case. It's possible to declare unconstrained multidimensional arrays as well. For example:

Listing 794: multidimensional_arrays_decl.ads

```

1 package Multidimensional_Arrays_Decl is
2
3     type F1 is array (Positive range <>) of Float;
4     type F2 is array (Positive range <>,
5                       Positive range <>) of Float;
6     type F3 is array (Positive range <>,
7                       Positive range <>,
8                       Positive range <>) of Float;
9
10 end Multidimensional_Arrays_Decl;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↳ Unconstrained_Multidimensional_Arrays
MD5: 8637e93db355fddafa3ffa5ce453a0e1

¹²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

¹²¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-6-2.html>

Here, we're declaring the one-dimensional type F1, the two-dimensional type F2 and the three-dimensional type F3.

As is the case with unidimensional arrays, we must specify the bounds when declaring objects of unconstrained multidimensional array types:

Listing 795: show_multidimensional_arrays.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Multidimensional_Arrays_Decl;
4 use Multidimensional_Arrays_Decl;
5
6 procedure Show_Multidimensional_Arrays is
7   A1 : F1 (1 .. 2);
8   A2 : F2 (1 .. 4, 10 .. 20);
9   A3 : F3 (2 .. 3, 1 .. 5, 1 .. 2);
10 begin
11   Put_Line ("A1'Length (1): "
12             & A1'Length (1)'Image);
13   Put_Line ("A2'Length (1): "
14             & A2'Length (1)'Image);
15   Put_Line ("A2'Length (2): "
16             & A2'Length (2)'Image);
17   Put_Line ("A3'Length (1): "
18             & A3'Length (1)'Image);
19   Put_Line ("A3'Length (2): "
20             & A3'Length (2)'Image);
21   Put_Line ("A3'Length (3): "
22             & A3'Length (3)'Image);
23 end Show_Multidimensional_Arrays;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Multidimensional_Arrays.
↳ Unconstrained_Multidimensional_Arrays
MD5: 9fb007abbfe238345d80cb315bb834c9

Build output

```
show_multidimensional_arrays.adb:7:04: warning: variable "A1" is read but never
↳ assigned [-gnatwv]
show_multidimensional_arrays.adb:8:04: warning: variable "A2" is read but never
↳ assigned [-gnatwv]
show_multidimensional_arrays.adb:9:04: warning: variable "A3" is read but never
↳ assigned [-gnatwv]
```

Runtime output

```
A1'Length (1): 2
A2'Length (1): 4
A2'Length (2): 11
A3'Length (1): 2
A3'Length (2): 5
A3'Length (3): 2
```

Arrays of arrays

It's important to distinguish between multidimensional arrays and arrays of arrays. Both are supported in Ada, but they're very distinct from each other. We can create an array of an array by first specifying a one-dimensional array type T1, and then specifying another one-dimensional array type T2 where each component of T2 is of T1 type:

Listing 796: array_of_arrays_decl.ads

```

1 package Array_Of_Arrays_Decl is
2
3   type T1 is
4     array (Positive range <>) of Float;
5
6   type T2 is
7     array (Positive range <>) of T1 (1 .. 10);
8     --           ^^^^^^^
9     --           bounds must be set!
10
11 end Array_Of_Arrays_Decl;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Of_Arrays.Array_Of_Arrays
MD5: fd67739bb21f202615180aa02f5284aa

Note that, in the declaration of T2, we must set the bounds for the T1 type. This is a major difference to multidimensional arrays, which allow for unconstrained ranges in multiple dimensions.

We can rewrite the previous application for measurements using arrays of arrays. This is the adapted code:

Listing 797: measurement_defs.ads

```

1 package Measurement_Defs is
2
3   type Days is
4     (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
5
6   type Hours is range 0 .. 11;
7
8   subtype Measurement is Float;
9
10  type Hourly_Measurements is
11    array (Hours) of Measurement;
12
13  type Measurements is
14    array (Days) of Hourly_Measurements;
15
16  procedure Show_Indices (M : Measurements);
17
18  procedure Show_Values (M : Measurements);
19
20  procedure Reset (M : out Measurements);
21
22 end Measurement_Defs;

```

Listing 798: measurement_defs.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 package body Measurement_Defs is
4
5   procedure Show_Indices (M : Measurements) is
6   begin
7     Put_Line ("---- Indices ----");
8

```

(continues on next page)

(continued from previous page)

```

9      for D in M'Range loop
10         Put (D'Image & " ");
11
12         for H in M (D)'First ..
13             M (D)'Last - 1
14         loop
15             Put (H'Image & " ");
16         end loop;
17         Put_Line (M (D)'Last'Image);
18     end loop;
19 end Show_Indices;
20
21 procedure Show_Values (M : Measurements) is
22     package H_IO is
23         new Ada.Text_IO.Integer_IO (Hours);
24     package M_IO is
25         new Ada.Text_IO.Float_IO (Measurement);
26
27     procedure Set_IO_Defaults is
28     begin
29         H_IO.Default_Width := 5;
30
31         M_IO.Default_Fore := 1;
32         M_IO.Default_Aft  := 2;
33         M_IO.Default_Exp  := 0;
34     end Set_IO_Defaults;
35 begin
36     Set_IO_Defaults;
37
38     Put_Line ("---- Values ----");
39     Put (" ");
40     for H in M (M'First)'Range loop
41         H_IO.Put (H);
42     end loop;
43     New_Line;
44
45     for D in M'Range loop
46         Put (D'Image & " ");
47
48         for H in M (D)'Range loop
49             M_IO.Put (M (D) (H));
50             Put (" ");
51         end loop;
52         New_Line;
53     end loop;
54 end Show_Values;
55
56 procedure Reset (M : out Measurements) is
57 begin
58     M := (others => (others => 0.0));
59 end Reset;
60
61 end Measurement_Defs;

```

Listing 799: show_measurements.adb

```

1 with Measurement_Defs; use Measurement_Defs;
2
3 procedure Show_Measurements is
4     M : Measurements;
5 begin

```

(continues on next page)

(continued from previous page)

```

6   Reset (M);
7   Show_Indices (M);
8   Show_Values (M);
9   end Show_Measurements;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Array_Of_Arrays.Multidimensional_Measurements
MD5: 5cb66bbb1890787b7c023406b2cafb4d

Runtime output

```

---- Indices ----
MON 0  1  2  3  4  5  6  7  8  9 10 11
TUE 0  1  2  3  4  5  6  7  8  9 10 11
WED 0  1  2  3  4  5  6  7  8  9 10 11
THU 0  1  2  3  4  5  6  7  8  9 10 11
FRI 0  1  2  3  4  5  6  7  8  9 10 11
SAT 0  1  2  3  4  5  6  7  8  9 10 11
SUN 0  1  2  3  4  5  6  7  8  9 10 11
---- Values ----
      0      1      2      3      4      5      6      7      8      9     10     11
MON 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
TUE 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
WED 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
THU 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
FRI 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SAT 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
SUN 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00

```

Again, we recommend that you spend some time analyzing this example and comparing it to the previous version that uses multidimensional arrays. Also, we'd like to highlight the following aspects:

- We access a value from an array of arrays by specifying the index of each array separately. For example: `M (D) (H)` allows us to access the value on day `D` and hour `H` from the array of arrays `M`.
- To loop over an array of arrays `M`, we write `for D in M'Range loop` for the first level of `M` and `for H in M (D)'Range loop` for the second level of `M`.
- Resetting all values of an array of arrays is very similar to how we do it for multidimensional arrays. In fact, we can still use an aggregate with this form: `(others => (others => 0.0))`.

25.6.3 Derived array types and array subtypes

Derived array types

As expected, we can derive from array types by declaring a new type. Let's see a couple of examples based on the `Measurement_Defs` package from previous sections:

Listing 800: `measurement_defs.ads`

```

1 package Measurement_Defs is
2
3   type Measurements is
4     array (Positive range <>) of Float;
5
6   --

```

(continues on next page)

(continued from previous page)

```

7  -- New array type:
8  --
9  type Measurements_Derived is
10     new Measurements;
11
12  --
13  -- New array type with
14  -- default component value:
15  --
16  type Measurements_Def30 is
17     new Measurements
18     with Default_Component_Value => 30.0;
19
20  --
21  -- New array type with constraints:
22  --
23  type Measurements_10 is
24     new Measurements (1 .. 10);
25
26 end Measurement_Defs;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.
↳ Derived_Arrays
MD5: aefef9b9a844ad820d7f16546b8ffa64

In this example, we're deriving `Measurements_Derived` from the `Measurements` type. In the case of the `Measurements_Def30` type, we're not only deriving from the `Measurements` type, but also setting the *default component value* (page 340) to 30.0. Finally, in the case of the `Measurements_10`, we're deriving from the `Measurements` type and *constraining the array type* (page 565) in the range from 1 to 10.

Let's use these types in a test application:

Listing 801: show_measurements.adb

```

1  with Measurement_Defs; use Measurement_Defs;
2
3  procedure Show_Measurements is
4      M1, M2 : Measurements (1 .. 10)
5              := (others => 0.0);
6
7      MD      : Measurements_Derived (1 .. 10);
8      MD2     : Measurements_Derived (1 .. 40);
9      MD10    : Measurements_10;
10 begin
11     M1 := M2;
12     -- ^^^^^
13     -- Assignment of arrays of
14     -- same type.
15
16     MD := Measurements_Derived (M1);
17     -- ~~~~~
18     -- Conversion to derived type for
19     -- the assignment.
20
21     MD10 := Measurements_10 (M1);
22     -- ~~~~~
23     -- Conversion to derived type for
24     -- the assignment.

```

(continues on next page)

(continued from previous page)

```

25
26     MD10 := Measurements_10 (MD);
27     MD10 := Measurements_10 (MD2 (1 .. 10));
28 end Show_Measurements;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.
↳ Derived_Arrays
MD5: ce37a9c17eb9e1bb3931cca82852b54a

Build output

```

show_measurements.adb:8:04: warning: variable "MD2" is read but never assigned [-
↳ gnatwv]

```

As illustrated by this example, we can assign objects of different array types, provided that we perform the appropriate type conversions and make sure that the bounds match.

Array subtypes

Naturally, we can also declare subtypes of array types. For example:

Listing 802: measurement_defs.ads

```

1 package Measurement_Defs is
2
3     type Measurements is
4         array (Positive range <>) of Float;
5
6     --
7     -- Simple subtype declaration:
8     --
9     subtype Measurements_Sub is Measurements;
10
11     --
12     -- Subtype with constraints:
13     --
14     subtype Measurements_10 is
15         Measurements (1 .. 10);
16
17     --
18     -- Subtype with dynamic predicate
19     -- (array can only have 20 components
20     -- at most):
21     --
22     subtype Measurements_Max_20 is Measurements
23         with Dynamic_Predicate =>
24             Measurements_Max_20'Length <= 20;
25
26     --
27     -- Subtype with constraints and
28     -- dynamic predicate (first element
29     -- must be 2.0).
30     --
31     subtype Measurements_First_Two is
32         Measurements (1 .. 10)
33         with Dynamic_Predicate =>
34             Measurements_First_Two (1) = 2.0;
35
36 end Measurement_Defs;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.Array_Subtypes
 MD5: fa03c836111aa2df223a38a5d04d18bc

Here, we're declaring subtypes of the Measurements type. For example, Measurements_Sub is a *simple* subtype of Measurements type. In the case of the Measurements_10 subtype, we're constraining the type to a range from 1 to 10.

For the Measurements_Max_20 subtype, we're specifying — via a dynamic predicate — that arrays of this subtype can only have 20 components at most. Finally, for the Measurements_First_Two subtype, we're constraining the type to a range from 1 to 10 and requiring that the first component must have a value of 2.0.

Note that we cannot set the default component value for array subtypes — only type declarations are allowed to use that facility.

Let's use these subtypes in a test application:

Listing 803: show_measurements.adb

```

1  with Measurement_Defs; use Measurement_Defs;
2
3  procedure Show_Measurements is
4      M1, M2 : Measurements (1 .. 10)
5          := (others => 0.0);
6      MS     : Measurements_Sub (1 .. 10);
7      MD10   : Measurements_10;
8      M_Max20 : Measurements_Max_20 (1 .. 40);
9      M_F2    : Measurements_First_Two;
10  begin
11      MS      := M1;
12      MD10    := M1;
13
14      M_Max20 := (others => 0.0); -- ERROR!
15
16      MD10 (1) := 4.0;
17      M_F2    := MD10;          -- ERROR!
18  end Show_Measurements;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Arrays.Derived_Arrays_And_Subtypes.Array_Subtypes
 MD5: 003ddaab65d8c163302811abd7889745

Runtime output

```

raised ADA.ASSERTIONS.ASSERTION_ERROR : Dynamic_Predicate failed at show_
measurements.adb:14

```

As expected, assignments to objects with different subtypes — but with the same parent type — work fine without conversion. The assignment to M_Max_20 fails because of the predicate failure: the predicate requires that the length be 20 at most, and it's 40 in this case. Also, the assignment to M_F2 fails because the predicate requires that the first element must be set to 2.0, and MD10 (1) has the value 4.0.

25.7 Strings

25.7.1 Character and String Literals

So far, we've already seen many examples of string literals — both in the *Introduction to Ada* (page 5) course and in the present course. In this section, we define them once more and discuss a couple of details about them.

Character Literals

A character literal is simply a character between apostrophes (or *single quotation marks*). For example:

Listing 804: show_character_literals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Character_Literals is
4   C : Character := 'a';
5   --           ^^^
6   --           Character literal
7 begin
8   Put_Line ("Character : " & C);
9 end Show_Character_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.
 ↪Character_Literals
 MD5: e9bf0dee97b4c6d52937316e7f285f48

Runtime output

```
Character : a
```

In this example, we initialize the character variable C with the character literal 'a'.

String Literals

A string literal is simply a collection of characters between quotation marks. For example:

Listing 805: show_simple_string_literals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_String_Literals is
4   S1 : String := "Hello";
5   --         ^^^^^^
6   --         String literal
7
8   S2 : String := "World";
9   --         ^^^^^^
10  --         String literal
11 begin
12   Put_Line (S1 & " " & S2);
13 end Show_Simple_String_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.Simple_String_Literals
MD5: a19bfa1ab6048f8cad9858d57b9f21e1

Runtime output

```
Hello World
```

In this example, "Hello" and "World" are string literals.

String literals with quotation

If you want to include a quotation mark in a string literal, you have to write `"` (inside that string literal):

Listing 806: show_string_literals_with_quotes.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_String_Literals_With_Quotes is
4   S1 : String := "Hello";
5   S2 : String := "World";
6 begin
7   Put_Line ("  " & S1
8     --      ^^
9     --      Quotation marks
10    & " " & S2 & "  " );
11   --      ^^
12   --      Quotation marks
13
14   Put_Line ("Hello World!");
15   --      ^^
16   --      Quotation marks
17
18   Put_Line ("");
19   --      ^^
20   --      Quotation marks
21 end Show_String_Literals_With_Quotes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.String_Literals_With_Quotes
MD5: 97752e289b5f58f98920407f5dedc1fb

Runtime output

```
" Hello World "
"Hello World!"
"
```

In this example, we display `" Hello World "` to the user by adding quotation marks to the concatenated strings in the call to `Put_Line`.

Note that the three quotation marks at the beginning of `""Hello World!""` consist of the quotation mark that indicate the beginning of the string literal and the two quotation marks that represent a single quotation mark inside the string literal. (The same thing happens at the end of this string literal, but in reverse.) This string literal is displayed as `"Hello World!"` to the user.

Finally, the string literal `""` is displayed as `"` to the user.

Empty string literals

An empty string is represented by quotation marks without characters in between: `""`. For example:

Listing 807: show_empty_string_literals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Empty_String_Literals is
4   S1 : String := "";
5   S2 : String (1 .. 0) := "";
6 begin
7   Put_Line (S1);
8   Put_Line (S2);
9   Put_Line ("");
10 end Show_Empty_String_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Character_String_Literals.Empty_String_Literals
 MD5: f2f7c47784f1053665db9499cb6b53d8

Runtime output

Note that an empty string is an array of characters without any components. This is made explicit by the declaration of S2. Here, by using the range `1 .. 0`, we're declaring an empty array.

In other languages

In C, an empty string still contains a single character: the null character (`\0`). In Ada, however, an empty string doesn't have any characters.

In the Ada Reference Manual

- 2.5 Character Literals¹²²
- 2.6 String Literals¹²³

25.7.2 Wide and Wide-Wide Strings

We've seen many source-code examples so far that includes strings. In most of them, we were using the standard string type: `String`. This type is useful for the common use-case of displaying messages or dealing with information in plain English. Here, we define "plain English" as the use of the language that avoids French accents or German umlaut, for example, and doesn't make use of any characters in non-Latin alphabets.

There are two additional string types in Ada: `Wide_String`, and `Wide_Wide_String`. These types are particularly important when dealing with textual information in non-standard English, or in various other languages, non-Latin alphabets and special symbols.

¹²² <http://www.ada-auth.org/standards/22rm/html/RM-2-5.html>

¹²³ <http://www.ada-auth.org/standards/22rm/html/RM-2-6.html>

These string types use different bit widths for their characters. This becomes more apparent when looking at the type definitions:

```
type String is
  array (Positive range <>) of Character;

type Wide_String is
  array (Positive range <>) of Wide_Character;

type Wide_Wide_String is
  array (Positive range <>) of
    Wide_Wide_Character;
```

The following table shows the typical bit-width of each character of the string types:

Character Type	Width
Character	8 bits
Wide_Character	16 bits
Wide_Wide_Character	32 bits

We can see that when running this example:

Listing 808: show_wide_char_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Wide_Char_Types is
4 begin
5   Put_Line ("Character'Size:      ")
6     & Integer'Image
7     (Character'Size));
8   Put_Line ("Wide_Character'Size:  ")
9     & Integer'Image
10    (Wide_Character'Size));
11  Put_Line ("Wide_Wide_Character'Size: ")
12    & Integer'Image
13    (Wide_Wide_Character'Size));
14 end Show_Wide_Char_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Strings.Wide_Char_Types
 MD5: a0e9fb9e8d43e9fa707dc8c57f7562f8

Runtime output

```
Character'Size:      8
Wide_Character'Size: 16
Wide_Wide_Character'Size: 32
```

Let's look at another example, this time using wide strings:

Listing 809: show_wide_string_types.adb

```
1 with Ada.Text_IO;
2 with Ada.Wide_Text_IO;
3 with Ada.Wide_Wide_Text_IO;
4
```

(continues on next page)

(continued from previous page)

```

5  procedure Show_Wide_String_Types is
6      package TI    renames Ada.Text_IO;
7      package WTI   renames Ada.Wide_Text_IO;
8      package WWTI  renames Ada.Wide_Wide_Text_IO;
9
10     S    : constant String      := "hello";
11     WS   : constant Wide_String := "hello";
12     WWS  : constant Wide_Wide_String := "hello";
13 begin
14     TI.Put_Line ("String:      " & S);
15     TI.Put_Line ("Length:      "
16                 & Integer'Image (S'Length));
17     TI.Put_Line ("Size:        "
18                 & Integer'Image (S'Size));
19     TI.Put_Line ("Component_Size:  "
20                 & Integer'Image
21                 (S'Component_Size));
22     TI.Put_Line ("-----");
23
24     WTI.Put_Line ("Wide string:      " & WS);
25     TI.Put_Line ("Length:      "
26                 & Integer'Image (WS'Length));
27     TI.Put_Line ("Size:        "
28                 & Integer'Image (WS'Size));
29     TI.Put_Line ("Component_Size:  "
30                 & Integer'Image
31                 (WS'Component_Size));
32     TI.Put_Line ("-----");
33
34     WWTI.Put_Line ("Wide-wide string: " & WWS);
35     TI.Put_Line ("Length:      "
36                 & Integer'Image (WWS'Length));
37     TI.Put_Line ("Size:        "
38                 & Integer'Image (WWS'Size));
39     TI.Put_Line ("Component_Size:  "
40                 & Integer'Image
41                 (WWS'Component_Size));
42     TI.Put_Line ("-----");
43 end Show_Wide_String_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Wide_
 ↪String_Types
 MD5: 137816c6fd78add34287a72e45cf4fb7

Runtime output

```

String:      hello
Length:      5
Size:        40
Component_Size:  8
-----
Wide string:  hello
Length:      5
Size:        80
Component_Size: 16
-----
Wide-wide string: hello
Length:      5
Size:        160

```

(continues on next page)

(continued from previous page)

```
Component_Size:    32
-----
```

Here, all strings (S, WS and WWS) have the same length of 5 characters. However, the size of each character is different — thus, each string has a different overall size.

The recommendation is to use the **String** type when the textual information you're processing is in standard English. In case any kind of internationalization is needed, using `Wide_Wide_String` is probably the best choice, as it covers all possible use-cases.

In the Ada Reference Manual

- [3.6.3 String Types](#)¹²⁴

Text I/O

Note that, in the previous example, we were using different versions of the `Ada.Text_IO` package depending on the string type we were using:

- `Ada.Text_IO` for objects of **String** type,
- `Ada.Wide_Text_IO` for objects of **Wide_String** type,
- `Ada.Wide_Wide_Text_IO` for objects of `Wide_Wide_String` type.

In that example, we were also using package renaming to differentiate among those packages.

Similarly, there are different versions of text I/O packages for individual types. For example, if we want to display the value of a **Long_Integer** variable based on the `Wide_Wide_String` type, we can select the `Ada.Long_Integer_Wide_Wide_Text_IO` package. In fact, the list of packages resulting from the combination of those types is quite long:

¹²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-6-3.html>

Scalar Type	Text I/O Packages
Integer	• Ada.Integer_Text_IO • Ada.Integer_Wide_Text_IO • Ada.Integer_Wide_Wide_Text_IO
Long_Integer	• Ada.Long_Integer_Text_IO • Ada.Long_Integer_Wide_Text_IO • Ada.Long_Integer_Wide_Wide_Text_IO
Long_Long_Integer	• Ada.Long_Long_Integer_Text_IO • Ada.Long_Long_Integer_Wide_Text_IO • Ada.Long_Long_Integer_Wide_Wide_Text_IO
Float	• Ada.Float_Text_IO • Ada.Float_Wide_Text_IO • Ada.Float_Wide_Wide_Text_IO
Long_Float	• Ada.Long_Float_Text_IO • Ada.Long_Float_Wide_Text_IO • Ada.Long_Float_Wide_Wide_Text_IO
Long_Long_Float	• Ada.Long_Long_Float_Text_IO • Ada.Long_Long_Float_Wide_Text_IO • Ada.Long_Long_Float_Wide_Wide_Text_IO

Also, there are different versions of the generic packages Integer_IO and Float_IO:

Scalar Type	Text I/O Packages
Integer types	• Ada.Text_IO.Integer_IO • Ada.Wide_Text_IO.Integer_IO • Ada.Wide_Wide_Text_IO.Integer_IO
Real types	• Ada.Text_IO.Float_IO • Ada.Wide_Text_IO.Float_IO • Ada.Wide_Wide_Text_IO.Float_IO

In the Ada Reference Manual

- A.10 Text Input-Output¹²⁵
- A.10.1 The Package Text_IO¹²⁶
- A.10.8 Input-Output for Integer Types¹²⁷

- [A.10.9 Input-Output for Real Types](#)¹²⁸
- [A.11 Wide Text Input-Output and Wide Wide Text Input-Output](#)¹²⁹

Wide and Wide-Wide String Handling

As we've just seen, we have different versions of the `Ada.Text_IO` package. The same applies to string handling packages. As we've seen in the [Introduction to Ada course](#) (page 241), we can use the `Ada.Strings.Fixed` and `Ada.Strings.Maps` packages for string handling. For other formats, we have these packages:

- `Ada.Strings.Wide_Fixed`,
- `Ada.Strings.Wide_Wide_Fixed`,
- `Ada.Strings.Wide_Maps`,
- `Ada.Strings.Wide_Wide_Maps`.

Let's look at [this example](#) (page 242) from the Introduction to Ada course, which we adapted for wide-wide strings:

Listing 810: `show_find_words.adb`

```
1  with Ada.Strings; use Ada.Strings;
2
3  with Ada.Strings.Wide_Wide_Fixed;
4  use  Ada.Strings.Wide_Wide_Fixed;
5
6  with Ada.Strings.Wide_Wide_Maps;
7  use  Ada.Strings.Wide_Wide_Maps;
8
9  with Ada.Wide_Wide_Text_IO;
10 use  Ada.Wide_Wide_Text_IO;
11
12 procedure Show_Find_Words is
13
14     S    : constant Wide_Wide_String :=
15           "Hello" & 3 * " World";
16     F    : Positive;
17     L    : Natural;
18     I    : Natural := 1;
19
20     Whitespace : constant
21       Wide_Wide_Character_Set :=
22       To_Set (' ');
23 begin
24     Put_Line ("String: " & S);
25     Put_Line ("String length: "
26             & Integer'Wide_Wide_Image
27             (S'Length));
28
29     while I in S'Range loop
30         Find-Token
31           (Source => S,
32            Set    => Whitespace,
33            From   => I,
```

(continues on next page)

¹²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-10.html>

¹²⁶ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-1.html>

¹²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-8.html>

¹²⁸ <http://www.ada-auth.org/standards/22rm/html/RM-A-10-9.html>

¹²⁹ <http://www.ada-auth.org/standards/22rm/html/RM-A-11.html>

(continued from previous page)

```

34         Test    => Outside,
35         First   => F,
36         Last    => L);
37
38     exit when L = 0;
39
40     Put_Line ("Found word instance at position "
41             & F'Wide_Wide_Image
42             & ": '" & S (F .. L) & "'");
43
44     I := L + 1;
45 end loop;
46
47 end Show_Find_Words;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Strings.Wide_Wide_
 ↳String_Handling
 MD5: 3b5a4d61e6dc5bd16e85f85580ad82ae

Runtime output

```

String: Hello World World World
String length: 23
Found word instance at position 1: 'Hello'
Found word instance at position 7: 'World'
Found word instance at position 13: 'World'
Found word instance at position 19: 'World'

```

In this example, we're using the `Find-Token` procedure to find the words from the phrase stored in the `S` constant. All the operations we're using here are similar to the ones for **String** type, but making use of the `Wide_Wide_String` type instead. (We talk about the `Wide_Wide_Image` attribute *later on* (page 607).)

In the Ada Reference Manual

- [A.4.6 String-Handling Sets and Mappings](#)¹³⁰
- [A.4.7 Wide_String Handling](#)¹³¹
- [A.4.8 Wide_Wide_String Handling](#)¹³²

Bounded and Unbounded Wide and Wide-Wide Strings

We've seen in the Introduction to Ada course that other kinds of **String** types are available. For example, we can use *bounded* (page 246) and *unbounded strings* (page 249) — those correspond to the `Bounded_String` and `Unbounded_String` types.

Those kinds of string types are available for **Wide_String**, and `Wide_Wide_String`. The following table shows the available types and corresponding packages:

¹³⁰ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-6.html>

¹³¹ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html>

¹³² <http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html>

Type	Package
Bounded_Wide_String	Ada.Strings.Wide_Bounded
Bounded_Wide_Wide_String	Ada.Strings.Wide_Wide_Bounded
Unbounded_Wide_String	Ada.Strings.Wide_Unbounded
Unbounded_Wide_Wide_String	Ada.Strings.Wide_Wide_Unbounded

The same applies to text I/O for those strings. For the standard case, we have `Ada.Text_IO.Bounded_IO` for the `Bounded_String` type and `Ada.Text_IO.Unbounded_IO` for the `Unbounded_String` type.

For wider string types, we have:

Type	Text I/O Package
Bounded_Wide_String	Ada.Wide_Text_IO.Wide_Bounded_IO
Bounded_Wide_Wide_String	Ada.Wide_Wide_Text_IO.Wide_Wide_Bounded_IO
Unbounded_Wide_String	Ada.Wide_Text_IO.Wide_Unbounded_IO
Unbounded_Wide_Wide_String	Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO

Let's look at a simple example:

Listing 811: `show_unbounded_wide_wide_string.adb`

```

1 with Ada.Strings.Wide_Wide_Unbounded;
2 use  Ada.Strings.Wide_Wide_Unbounded;
3
4 with Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;
5 use  Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO;
6
7 procedure Show_Unbounded_Wide_Wide_String is
8   S : Unbounded_Wide_Wide_String
9     := To_Unbounded_Wide_Wide_String ("Hello");
10 begin
11   S := S & Wide_Wide_String'(" hello");
12   Put_Line ("Unbounded wide-wide string: " & S);
13 end Show_Unbounded_Wide_Wide_String;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Strings.Wide_Wide-Wide_Strings.Unbounded_Wide_Wide_String`
MD5: `0d369270e2408b3f1cc8284c13fca806`

Runtime output

Unbounded wide-wide string: Hello hello

In this example, we're declaring a variable `S` and initializing it with the word "Hello." Then, we're concatenating it with " hello" and displaying it. All the operations we're using here are similar to the ones for `Unbounded_String` type, but they've been adapted for the `Unbounded_Wide_Wide_String` type.

In the Ada Reference Manual

- [A.4.7 Wide_String Handling](#)¹³³

- A.4.8 Wide_Wide_String Handling¹³⁴
- A.11 Wide Text Input-Output and Wide Wide Text Input-Output¹³⁵

25.7.3 String Encoding

Unicode is one of the most widespread standards for encoding writing systems other than the Latin alphabet. It defines a format called [Unicode Transformation Format \(UTF\)](#)¹³⁶ in various versions, which vary according to the underlying precision, support for backwards-compatibility and other requirements.

i In the Ada Reference Manual

- A.4.11 String Encoding¹³⁷

UTF-8 encoding and decoding

A common UTF format is UTF-8, which encodes strings using up to four (8-bit) bytes and is backwards-compatible with the ASCII format. While encoding of ASCII characters requires only one byte, Chinese characters require three bytes, for example.

In Ada applications, UTF-8 strings are indicated by using the `UTF_8_String` from the `Ada.Strings.UTF_Encoding` package. In order to encode from and to UTF-8 strings, we can use the `Encode` and `Decode` functions. Those functions are specified in the child packages of the `Ada.Strings.UTF_Encoding` package. We select the appropriate child package depending on the string type we're using, as you can see in the following table:

Child Package of <code>Ada.Strings.UTF_Encoding</code>	Convert from / to
<code>.Strings</code>	String type
<code>.Wide_Strings</code>	Wide_String type
<code>.Wide_Wide_Strings</code>	<code>Wide_Wide_String</code> type

Let's look at an example:

Listing 812: `show_ww_utf_string.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  with Ada.Strings.Wide_Wide_Unbounded;
10 use  Ada.Strings.Wide_Wide_Unbounded;
11
12 procedure Show_WW_UTF_String is
13
14     function To_UWWS
```

(continues on next page)

¹³³ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-7.html>

¹³⁴ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-8.html>

¹³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-11.html>

¹³⁶ https://unicode.org/faq/utf_bom.html#gen2

¹³⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-11.html>

(continued from previous page)

```

15     (Source : Wide_Wide_String)
16     return Unbounded_Wide_Wide_String
17     renames To_Unbounded_Wide_Wide_String;
18
19 function To_WWS
20 (Source : Unbounded_Wide_Wide_String)
21 return Wide_Wide_String
22 renames To_Wide_Wide_String;
23
24 Hello_World_Arabic : constant
25   UTF_8_String := "عالم يا مرحبا";
26 WWS_Hello_World_Arabic : constant
27   Wide_Wide_String :=
28     Decode (Hello_World_Arabic);
29
30 UWWS : Unbounded_Wide_Wide_String;
31 begin
32   UWWS := "Hello World: "
33     & To_UWWS (WWS_Hello_World_Arabic);
34
35 Show_WW_String : declare
36   WWS : constant Wide_Wide_String :=
37     To_WWS (UWWS);
38 begin
39   Put_Line ("Wide_Wide_String Length: "
40     & WWS'Length'Image);
41   Put_Line ("Wide_Wide_String Size: "
42     & WWS'Size'Image);
43 end Show_WW_String;
44
45 Put_Line
46   ("-----");
47 Put_Line
48   ("Converting Wide_Wide_String to UTF-8...");
49
50 Show_UTF_8_String : declare
51   S_UTF_8 : constant UTF_8_String :=
52     Encode (To_WWS (UWWS));
53 begin
54   Put_Line ("UTF-8 String: "
55     & S_UTF_8);
56   Put_Line ("UTF-8 String Length: "
57     & S_UTF_8'Length'Image);
58   Put_Line ("UTF-8 String Size: "
59     & S_UTF_8'Size'Image);
60 end Show_UTF_8_String;
61
62 end Show_WW_UTF_String;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WW_UTF_String
 MD5: cecfb420bb804f42e7a65b793abcbef5

Runtime output

```

Wide_Wide_String Length:  26
Wide_Wide_String Size:   832
-----
Converting Wide_Wide_String to UTF-8...
UTF-8 String:           Hello World: عالم يا مرحبا

```

(continues on next page)

(continued from previous page)

```

UTF-8 String Length:  37
UTF-8 String Size:    296

```

In this application, we start by storing a string in Arabic in the `Hello_World_Arabic` constant. We then use the `Decode` function to convert that string from `UTF_8_String` type to `Wide_Wide_String` type — we store it in the `WWS_Hello_World_Arabic` constant.

We use a variable of type `Unbounded_Wide_Wide_String` (UWWS) to manipulate strings: we append the string in Arabic to the "Hello World: " string and store it in UWWS.

In the `Show_WW_String` block, we convert the string — stored in UWWS — from the `Unbounded_Wide_Wide_String` type to the `Wide_Wide_String` type and display the length and size of the string. We do something similar in the `Show_UTF_8_String` block, but there, we convert to the `UTF_8_String` type.

Also, in the `Show_UTF_8_String` block, we use the `Encode` function to convert that string from `Wide_Wide_String` type to then `UTF_8_String` type — we store it in the `S_UTF_8` constant.

UTF-8 size and length

As you can see when running the last code example from the previous subsection, we have different sizes and lengths depending on the string type:

String type	Size	Length
<code>Wide_Wide_String</code>	832	26
<code>UTF_8_String</code>	296	37

The size needed for storing the string when using the `Wide_Wide_String` type is bigger than the one when using the `UTF_8_String` type. This is expected, as the `Wide_Wide_String` uses 32-bit characters, while the `UTF_8_String` type uses 8-bit codes to store the string in a more efficient way (memory-wise).

The length of the string using the `Wide_Wide_String` type is equivalent to the number of symbols we have in the original string: 26 characters / symbols. When using UTF-8, however, we may need more 8-bit codes to represent one symbol from the original string, so we may end up with a length value that is bigger than the actual number of symbols from the original string — as it is the case in this source-code example.

This difference in sizes might not always be the case. In fact, the sizes match when encoding a symbol in UTF-8 that requires four 8-bit codes. For example:

Listing 813: `show_utf_8.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  procedure Show_UTF_8 is
10
11     Symbol_UTF_8 : constant UTF_8_String := "x";
12     Symbol_WWS   : constant Wide_Wide_String :=
13         Decode (Symbol_UTF_8);
14
15  begin

```

(continues on next page)

(continued from previous page)

```

16 Put_Line ("Wide_Wide_String Length: "
17           & Symbol_WWS'Length'Image);
18 Put_Line ("Wide_Wide_String Size: "
19           & Symbol_WWS'Size'Image);
20 Put_Line ("UTF-8 String Length: "
21           & Symbol_UTF_8'Length'Image);
22 Put_Line ("UTF-8 String Size: "
23           & Symbol_UTF_8'Size'Image);
24 New_Line;
25 Put_Line ("UTF-8 String: "
26           & Symbol_UTF_8);
27 end Show_UTF_8;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8
MD5: 67653dfd377f04b32421cf09b25939fe

Runtime output

```

Wide_Wide_String Length:  1
Wide_Wide_String Size:   32
UTF-8 String Length:     4
UTF-8 String Size:       32

UTF-8 String:            x

```

In this case, both strings — using the `Wide_Wide_String` type or the `UTF_8_String` type — have the same size: 32 bits. (Here, we're using the `x` symbol from the [Mathematical Alphanumeric Symbols block](#)¹³⁸, not the standard `"x"` from the [Basic Latin block](#)¹³⁹.)

UTF-16 encoding and decoding

So far, we've discussed the UTF-8 encoding scheme. However, other encoding schemes exist and are supported as well. In fact, the `Ada.Strings.UTF_Encoding` package defines three encoding schemes:

```

type Encoding_Scheme is (UTF_8,
                        UTF_16BE,
                        UTF_16LE);

```

For example, instead of using UTF-8 encoding, we can use UTF-16 encoding — either in the big-endian or in the little-endian version. To convert between UTF-8 and UTF-16 encoding schemes, we can make use of the conversion functions from the `Ada.Strings.UTF_Encoding.Conversions` package.

To declare a UTF-16 encoded string, we can use one of the following data types:

- the 8-bit-character based `UTF_String` type, or
- the 16-bit-character based `UTF_16_Wide_String` type.

When using the 8-bit version, though, we have to specify the input and output schemes when converting between UTF-8 and UTF-16 encoding schemes.

Let's see a code example that makes use of both `UTF_String` and `UTF_16_Wide_String` types:

¹³⁸ https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols

¹³⁹ [https://en.wikipedia.org/wiki/Basic_Latin_\(Unicode_block\)](https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block))

Listing 814: show_utf16_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Conversions;
7  use  Ada.Strings.UTF_Encoding.Conversions;
8
9  procedure Show_UTF16_Types is
10     Symbols_UTF_8 : constant
11         UTF_8_String := "♥♪";
12
13     Symbols_UTF_16 : constant
14         UTF_16_Wide_String :=
15             Convert (Symbols_UTF_8);
16     -- ^ Calling Convert for UTF_8_String
17     --   to UTF_16_Wide_String conversion.
18
19     Symbols_UTF_16BE : constant
20         UTF_String :=
21             Convert (Item          => Symbols_UTF_8,
22                     Input_Scheme => UTF_8,
23                     Output_Scheme => UTF_16BE);
24     -- ^ Calling Convert for UTF_8_String
25     --   to UTF_String conversion in UTF-16BE
26     --   encoding.
27 begin
28     Put_Line ("UTF_8_String:      "
29             & Symbols_UTF_8);
30
31     Put_Line ("UTF_16_Wide_String:  "
32             & Convert (Symbols_UTF_16));
33     -- ^ Calling Convert for
34     --   the UTF_16_Wide_String to
35     --   UTF_8_String conversion.
36
37     Put_Line
38         ("UTF_String / UTF_16BE: "
39         & Convert
40             (Item          => Symbols_UTF_16BE,
41              Input_Scheme => UTF_16BE,
42              Output_Scheme => UTF_8));
43 end Show_UTF16_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_16_Types
MD5: 905e20e83a6199fdc91a6b15bb71bb01

Runtime output

```

UTF_8_String:      ♥♪
UTF_16_Wide_String: ♥♪
UTF_String / UTF_16BE: ♥♪

```

In this example, we're declaring a UTF-8 encoded string and storing it in the `Symbols_UTF_8` constant. Then, we're calling the `Convert` functions to convert between UTF-8 and UTF-16 encoding schemes. We're using two versions of this function:

- the `Convert` function that returns an object of `UTF_16_Wide_String` type for an input

of UTF_8_String type, and

- the Convert function that returns an object of UTF_String type for an input of UTF_8_String type.
 - In this case, we need to specify the input and output schemes (see Input_Scheme and Output_Scheme parameters in the code example).

Previously, we've seen that the Ada.Strings.UTF_Encoding.Wide_Wide_Strings package offers functions to convert between UTF-8 and the Wide_Wide_String type. The same kind of conversion functions exist for UTF-16 strings as well. Let's look at this code example:

Listing 815: show_ww_utf16_string.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  with Ada.Strings.UTF_Encoding.Conversions;
10 use  Ada.Strings.UTF_Encoding.Conversions;
11
12 procedure Show_WW_UTF16_String is
13   Symbols_UTF_16 : constant
14     UTF_16_Wide_String :=
15       Wide_Character'Val (16#2665#) &
16       Wide_Character'Val (16#266B#);
17   -- ^ Calling Wide_Character'Val
18   -- to specify the UTF-16 BE code
19   -- for "♥" and "♪".
20
21   Symbols_WWS : constant
22     Wide_Wide_String :=
23       Decode (Symbols_UTF_16);
24   -- ^ Calling Decode for UTF_16_Wide_String
25   -- to Wide_Wide_String conversion.
26 begin
27   Put_Line ("UTF_16_Wide_String: "
28     & Convert (Symbols_UTF_16));
29   -- ^ Calling Convert for the
30   -- UTF_16_Wide_String to
31   -- UTF_8_String conversion.
32
33   Put_Line ("Wide_Wide_String: "
34     & Encode (Symbols_WWS));
35   -- ^ Calling Encode for the
36   -- Wide_Wide_String to
37   -- UTF_8_String conversion.
38 end Show_WW_UTF16_String;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WW_UTF_16_String
MD5: 900af8f5c6aad7303c3e49c1c4a68d73

Runtime output

```
UTF_16_Wide_String: ♥♪
Wide_Wide_String:   ♥♪
```

In this example, we're calling the `Wide_Character'Val` function to specify the UTF-16 BE

code of the "♥" and "♪" symbols. We're then using the Decode function to convert between the UTF_16_Wide_String and the Wide_Wide_String types.

25.7.4 UTF-8 applications

In this section, we take a further look into UTF-8 encoding and some real-world applications. First, we discuss the use of UTF-8 encoding in source-code files. Then, we talk about parsing UTF-8 files using *wide-wide* strings.

UTF-8 encoding in source-code files

In the past, it was common to use different character sets in text files when writing in different (human) languages. By default, Ada source-code files are expected to use the Latin-1 coding, which is a 8-bit character set.

Nowadays, however, using UTF-8 coding for text files — including source-code files — is very common. If your Ada code only uses standard ASCII characters, but you're saving it in a UTF-8 coded file, there's no need to worry about character sets, as UTF-8 is backwards compatible with ASCII.

However, you might want to use Unicode symbols in your Ada source code to declare constants — as we did in the previous sections — and store the source code in a UTF-8 coded file. In this case, you need be careful about how this file is parsed by the compiler.

Let's look at this source-code example:

Listing 816: show_utf_8_strings.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use   Ada.Strings.UTF_Encoding;
5
6  procedure Show_UTF_8_Strings is
7
8      Symbols_UTF_8 : constant
9          UTF_8_String := "♥♪";
10
11  begin
12      Put_Line ("UTF_8_String: "
13              & Symbols_UTF_8);
14
15      Put_Line ("Length:      "
16              & Symbols_UTF_8'Length'Image);
17
18  end Show_UTF_8_Strings;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_Strings
MD5: fd1aaff161a33365d15adca5bea7b277

Runtime output

```
UTF_8_String: ♥♪
Length:      6
```

Here, we're using Unicode symbols to initialize the Symbols_UTF_8 constant of UTF_8_String type.

Now, let's assume this source-code example is stored in a UTF-8 coded file. Because the "♥♪" string makes use of non-ASCII Unicode symbols, representing this string in UTF-8 for-

mat will require more than 2 bytes. In fact, each one of those Unicode symbols requires 2 bytes to be encoded in UTF-8. (Keep in mind that Unicode symbols may require [between 1 to 4 bytes](https://en.wikipedia.org/wiki/UTF-8)¹⁴⁰ to be encoded in UTF-8 format.) Also, in this case, the UTF-8 encoding process is using two additional bytes. Therefore, the total length of the string is six, which matches what we see when running the `Show_UTF_8_Strings` procedure. In other words, the length of the `Symbols_UTF_8` string doesn't refer to those two characters ("♥♪") that we were using in the constant declaration, but the length of the encoded bytes in its UTF-8 representation.

The UTF-8 format is very useful for storing and transmitting texts. However, if we want to process Unicode symbols, it's probably better to use string types with 32-bit characters — such as `Wide_Wide_String`. For example, let's say we want to use the "♥♪" string again to initialize a constant of `Wide_Wide_String` type:

Listing 817: `show_wws_strings.adb`

```
1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO   renames Ada.Text_IO;
7     package WWTIO renames Ada.Wide_Wide_Text_IO;
8
9     Symbols_WWS : constant
10         Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                    & Symbols_WWS);
15
16     TIO.Put_Line ("Length: "
17                  & Symbols_WWS'Length'Image);
18
19 end Show_WWS_Strings;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String-Encoding.WWS_Strings_W8
MD5: 1e5e38e62b412de48d3fa4271bb48bf1

Runtime output

```
Wide_Wide_String: ♥♪
Length:          2
```

In this case, as mentioned above, if we store this source code in a text file using UTF-8 format, we need to ensure that the UTF-8 coded symbols are correctly interpreted by the compiler when it parses the text file. Otherwise, we might get unexpected behavior. (Interpreting the characters in UTF-8 format as Latin-1 format is certainly an example of what we want to avoid here.)

In the GNAT toolchain

You can use UTF-8 coding in your source-code file and initialize strings of 32-bit characters. However, as we just mentioned, you need to make sure that the UTF-8 coded symbols are correctly interpreted by the compiler when dealing with types such as `Wide_Wide_String`. For this case, GNAT offers the `-gnatW8` switch. Let's run the previous example using this switch:

¹⁴⁰ <https://en.wikipedia.org/wiki/UTF-8>

Listing 818: show_wws_strings.adb

```

1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO    renames Ada.Text_IO;
7     package WWTIO  renames Ada.Wide_Wide_Text_IO;
8
9     Symbols_WWS : constant
10         Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                     & Symbols_WWS);
15
16     TIO.Put_Line ("Length: "
17                  & Symbols_WWS'Length'Image);
18
19 end Show_WWS_Strings;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String-Encoding.WWS_Strings_W8
MD5: 1e5e38e62b412de48d3fa4271bb48bf1

Runtime output

```
Wide_Wide_String: ♥♪
Length:          2
```

Because the `Wide_Wide_String` type has 32-bit characters, we expect the length of the string to match the number of symbols that we're using. Indeed, when running the `Show_WWS_Strings` procedure, we see that the `Symbols_WWS` string has a length of two characters, which matches the number of characters of the `"♥♪"` string.

When we use the `-gnatw8` switch, GNAT converts the UTF-8-coded string (`"♥♪"`) to UTF-32 format, so we get two 32-bit characters. It then uses the UTF-32-coded string to initialize the `Symbols_WWS` string.

If we don't use the `-gnatw8` switch, however, we get wrong results. Let's look at the same example again without the switch:

Listing 819: show_wws_strings.adb

```

1 with Ada.Text_IO;
2 with Ada.Wide_Wide_Text_IO;
3
4 procedure Show_WWS_Strings is
5
6     package TIO    renames Ada.Text_IO;
7     package WWTIO  renames Ada.Wide_Wide_Text_IO;
8
9     Symbols_WWS : constant
10         Wide_Wide_String := "♥♪";
11
12 begin
13     WWTIO.Put_Line ("Wide_Wide_String: "
14                     & Symbols_WWS);
15
16     TIO.Put_Line ("Length: "
17                  & Symbols_WWS'Length'Image);
18
19 end Show_WWS_Strings;
```


Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.WWS_Strings_No_W8
 MD5: 1e5e38e62b412de48d3fa4271bb48bf1

Runtime output

Wide_Wide_String: ♥♪
 Length: 6

Now, the "♥♪" string is being interpreted as a string of six 8-bit characters. (In other words, the UTF-8-coded string isn't converted to the UTF-32 format.) Each of those 8-bit characters is then stored in a 32-bit character of the Wide_Wide_String type. This explains why the Show_WWS_Strings procedure reports a length of 6 components for the Symbols_WWS string.

Portability of UTF-8 in source-code files

In a previous code example, we were assuming that the format that we use for the source-code file is UTF-8. This allows us to simply use Unicode symbols directly in strings:

```
Symbol_UTF_8 : constant UTF_8_String := "★";
```

This approach, however, might not be portable. For example, if the compiler uses a different string encoding for source-code files, it might interpret that Unicode character as something else — or just throw a compilation error.

If you're afraid that format mismatches might happen in your compilation environment, you may want to write strings in your code in a completely portable fashion, which consists in entering the exact sequence of codes in bytes — using the **Character**'Val function — for the symbols you want to use.

We can reuse parts of the previous example and replace the UTF-8 character with the corresponding UTF-8 code:

Listing 820: show_utf_8.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.UTF_Encoding;
4 use Ada.Strings.UTF_Encoding;
5
6 procedure Show_UTF_8 is
7
8     Symbol_UTF_8 : constant
9         UTF_8_String :=
10             Character'Val (16#e2#)
11             & Character'Val (16#98#)
12             & Character'Val (16#85#);
13
14 begin
15     Put_Line ("UTF-8 String: "
16             & Symbol_UTF_8);
17 end Show_UTF_8;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8
 MD5: 8ff02bc1793c0c5ac1ff24f62941af73

Runtime output

UTF-8 String: ★

Here, we use a sequence of three calls to the **Character**'**Val**(code) function for the UTF-8 code that corresponds to the "★" symbol.

Parsing UTF-8 files for Wide-Wide-String processing

A typical use-case is to parse a text file in UTF-8 format and use *wide-wide* strings to process the lines of that file. Before we look at the implementation that does that, let's first write a procedure that generate a text file in UTF-8 format:

Listing 821: generate_utf_8_file.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  procedure Generate_UTF_8_File
7    (Output_File_Name : String)
8  is
9    F : File_Type;
10   begin
11     Create (F, Out_File, Output_File_Name);
12     Put_Line (F, UTF_8_String'("♥♪"));
13     Put_Line
14       (F,
15        UTF_8_String'("عالم يا مرحبا"));
16     Close (F);
17   end Generate_UTF_8_File;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
 Processing
 MD5: 58c7591796bc1348796afa6db6f64d22

Procedure Generate_UTF_8_File writes two strings with non-Latin characters into the UTF-8 file indicated by the Output_File_Name parameter.

In addition, let's implement an auxiliary procedure to display the individual characters of a *wide-wide* string:

Listing 822: put_line_utf_8_characters.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  procedure Put_Line_UTF_8_Characters
10    (WSS : Wide_Wide_String)
11  is
12    procedure Put_Complete_UTF_8_String
13      (WSS : Wide_Wide_String)
14    is
15      S_UTF_8 : constant UTF_8_String :=
16        Encode (WSS);
```

(continues on next page)

(continued from previous page)

```

17  begin
18      Put_Line ("STRING: " & S_UTF_8);
19      Put_Line ("Length: "
20                & WSS'Length'Image
21                & " characters");
22      New_Line;
23  end Put_Complete_UTF_8_String;
24
25  -- This is a wrapper function of the
26  -- Encode function for the
27  -- Wide_Wide_Character type:
28  function Encode (Item : Wide_Wide_Character)
29      return UTF_8_String
30  is
31      SC : constant Wide_Wide_String (1 .. 1)
32          := (1 => Item);
33      -- We need a 1-character string
34      -- for the call to Encode.
35  begin
36      return Encode (SC);
37  end Encode;
38
39  procedure Put_UTF_8_Characters
40      (WSS : Wide_Wide_String) is
41  begin
42      for I in WSS'Range loop
43          Put (I'Image & ": ");
44          Put (Encode (WSS (I)));
45          New_Line;
46      end loop;
47  end Put_UTF_8_Characters;
48
49  begin
50      Put_Complete_UTF_8_String (WSS);
51      Put_UTF_8_Characters (WSS);
52      Put_Line ("-----");
53  end Put_Line_UTF_8_Characters;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
 ↳ Processing
 MD5: 14fae1f2b1d3795f3cef244f60082fcc

Finally, let's look at a code example that parses an UTF-8 file:

Listing 823: show_utf_8.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Strings.UTF_Encoding;
4  use  Ada.Strings.UTF_Encoding;
5
6  with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7  use  Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9  with Generate_UTF_8_File;
10 with Put_Line_UTF_8_Characters;
11
12 procedure Show_UTF_8 is
13

```

(continues on next page)

(continued from previous page)

```

14  File_Name : constant String :=
15          "utf-8_test.txt";
16
17  procedure Read_UTF_8_File
18      (Input_File_Name : String)
19  is
20      F : File_Type;
21  begin
22      Open (F, In_File, Input_File_Name);
23
24      while not End_Of_File (F) loop
25          declare
26              S_UTF8 : constant UTF_8_String
27                  := Get_Line (F);
28              S      : constant Wide_Wide_String
29                  := Decode (S_UTF8);
30          begin
31              Put_Line_UTF_8_Chars (S);
32          end;
33      end loop;
34      Close (F);
35  end Read_UTF_8_File;
36
37  begin
38      Generate_UTF_8_File (File_Name);
39      Read_UTF_8_File (File_Name);
40  end Show_UTF_8;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
 ↳ Processing
 MD5: 512ad5ac7c6d5936735f017bfe629aa3

Runtime output

```

STRING: ♥♪
Length:  2 characters

1: ♥
2: ♪
-----
STRING: عالم يا مرحبا
Length:  13 characters

1: م
2: ر
3: ح
4: ب
5: ا
6:
7: ي
8: ا
9:
10: ع
11: ا
12: ج
13: م
-----

```

The Show_UTF_8 procedure first calls the Generate_UTF_8_File procedure to generate a text file in UTF-8 format, and then calls the nested Read_UTF_8_File procedure to read

from that file — this is done by reading the 8-bit UTF-8 encoded string and decoding it into a string of `Wide_Wide_String` type.

(Note that we call the auxiliary `Put_Line_UTF_8_Characters` procedure to display the characters of each line we read from the UTF-8 file.)

For completeness, we include the nested `Read_Write_UTF_8_File` procedure, which not only reads each line from a UTF-8 file, but also writes it into another UTF-8 file:

Listing 824: `show_utf_8.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.UTF_Encoding;
4 use Ada.Strings.UTF_Encoding;
5
6 with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
7 use Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
8
9 with Generate_UTF_8_File;
10 with Put_Line_UTF_8_Characters;
11
12 procedure Show_UTF_8 is
13
14     File_Name_In  : constant String :=
15                     "utf-8_test.txt";
16     File_Name_Out : constant String :=
17                     "utf-8_copy.txt";
18
19     procedure Read_Write_UTF_8_File
20         (Input_File_Name,
21          Output_File_Name : String)
22     is
23         F_In, F_Out : File_Type;
24     begin
25         Open (F_In, In_File, Input_File_Name);
26         Create (F_Out, Out_File, Output_File_Name);
27
28         while not End_Of_File (F_In) loop
29             declare
30                 S : constant Wide_Wide_String :=
31                     Decode (Get_Line (F_In));
32             begin
33                 Put_Line_UTF_8_Characters (S);
34                 Put_Line (F_Out, Encode (S));
35             end;
36         end loop;
37
38         Close (F_In);
39         Close (F_Out);
40     end Read_Write_UTF_8_File;
41
42 begin
43     Generate_UTF_8_File (File_Name_In);
44
45     Read_Write_UTF_8_File
46         (Input_File_Name => File_Name_In,
47          Output_File_Name => File_Name_Out);
48 end Show_UTF_8;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_`
(continues on next page)

(continued from previous page)

↳ Processing
MD5: 8cd13e8a565266fa5dd854ff6a34524c

Runtime output

```
STRING: ♥♪
Length:  2 characters

1: ♥
2: ♪
-----
STRING: عالم يا مرحبا
Length: 13 characters

1: م
2: ر
3: ح
4: ب
5: ا
6:
7: ي
8: ا
9:
10: ع
11: ا
12: ج
13: م
-----
```

In the nested `Read_Write_UTF_8_File` procedure, we see both `Decode` and `Encode` functions being called to convert from and to the `UTF_8_String` type, respectively.

In the GNAT toolchain

If we use the `-gnatW8` switch, which we mentioned *in a previous section* (page 596), the implementation of `Generate_UTF_8_File` and `Put_Line_UTF_8_Characters` must be adapted. In addition, we can simplify the implementation of the `Show_UTF_8` procedure, too. (Note, however, that the previous implementation, which makes use of the `Decode` and `Encode` functions, would work fine as well.)

Listing 825: `put_line_utf_8_characters.adb`

```
1 with Ada.Wide_Wide_Text_IO;
2 use  Ada.Wide_Wide_Text_IO;
3
4 procedure Put_Line_UTF_8_Characters
5   (WSS : Wide_Wide_String)
6 is
7   procedure Put_Complete_UTF_8_String
8     (WSS : Wide_Wide_String)
9   is
10    begin
11      Put_Line ("STRING: " & WSS);
12      Put_Line ("Length: "
13                & WSS'Length'Wide_Wide_Image
14                & " characters");
15      New_Line;
16    end Put_Complete_UTF_8_String;
17
18    procedure Put_UTF_8_Characters
19      (WSS : Wide_Wide_String)
```

```

20  is
21  begin
22      for I in WSS'Range loop
23          Put (I'Wide_Wide_Image & ": ");
24          Put (WSS (I));
25          New_Line;
26      end loop;
27  end Put_UTF_8_Characters;
28
29  begin
30      Put_Complete_UTF_8_String (WSS);
31      Put_UTF_8_Characters (WSS);
32      Put_Line ("-----");
33  end Put_Line_UTF_8_Characters;

```

Listing 826: generate_utf_8_file.adb

```

1  with Ada.Wide_Wide_Text_IO;
2  use  Ada.Wide_Wide_Text_IO;
3
4  procedure Generate_UTF_8_File
5      (Output_File_Name : String)
6  is
7      F : File_Type;
8  begin
9      Create (F, Out_File, Output_File_Name);
10     Put_Line (F, "♥️");
11     Put_Line (F, "عالم" يا مرحبا);
12     Close (F);
13 end Generate_UTF_8_File;

```

Listing 827: show_utf_8.adb

```

1  with Ada.Wide_Wide_Text_IO;
2  use  Ada.Wide_Wide_Text_IO;
3
4  with Generate_UTF_8_File;
5  with Put_Line_UTF_8_Characters;
6
7  procedure Show_UTF_8 is
8
9      File_Name_In  : constant String :=
10         "utf-8_test.txt";
11      File_Name_Out : constant String :=
12         "utf-8_copy.txt";
13
14  procedure Read_Write_UTF_8_File
15      (Input_File_Name,
16       Output_File_Name : String)
17  is
18      F_In, F_Out : File_Type;
19  begin
20      Open (F_In, In_File, Input_File_Name);
21      Create (F_Out, Out_File, Output_File_Name);
22
23      while not End_Of_File (F_In) loop
24          declare
25              S : constant Wide_Wide_String :=
26                 Get_Line (F_In);
27          begin
28              Put_Line_UTF_8_Characters (S);
29              Put_Line (F_Out, S);
30          end;

```

```

31     end loop;
32
33     Close (F_In);
34     Close (F_Out);
35 end Read_Write_UTF_8_File;
36
37 begin
38     Generate_UTF_8_File (File_Name_In);
39
40     Read_Write_UTF_8_File
41     (Input_File_Name => File_Name_In,
42      Output_File_Name => File_Name_Out);
43 end Show_UTF_8;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.String_Encoding.UTF_8_File_
 Processing
 MD5: 8eed924f6d661a0a62ecb4d94be7027

Runtime output

```

STRING: ♥♪
Length:  2 characters

1: ♥
2: ♪
-----
STRING: عالم يا مرحبا
Length:  13 characters

1: م
2: ر
3: ح
4: ب
5: ا
6:
7: ي
8: ا
9:
10: ع
11: ا
12: ل
13: م
-----

```

In this version of the code, we've removed all references to the `UTF_8_String` type — as well as the `Decode` and `Encode` functions that we were using to convert from and to this type. In this case, all UTF-8 processing happens directly using strings of `Wide_Wide_Strings` type.

25.7.5 Image attribute

Overview

In the *Introduction to Ada* (page 13) course, we've seen that the `Image` attribute returns a string that contains a textual representation of an object. For example, we write `Integer'Image (V)` to get a string for the integer variable `V`:

Listing 828: show_simple_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4   V : Integer;
5 begin
6   V := 10;
7   Put_Line ("V: " & Integer'Image (V));
8 end Show_Simple_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
MD5: e38f6f1a0808f12bd53c1f3cf4983353

Runtime output

V: 10

Naturally, we can use the Image attribute with other scalar types. For example:

Listing 829: show_simple_image.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4   type Status is (Unknown, Off, On);
5
6   V : Float;
7   S : Status;
8 begin
9   V := 10.0;
10  S := Unknown;
11
12  Put_Line ("V: " & Float'Image (V));
13  Put_Line ("S: " & Status'Image (S));
14 end Show_Simple_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
MD5: d3369518b610b7bf6c8dcefdecdb0c44

Runtime output

V: 1.00000E+01
S: UNKNOWN

In this example, we retrieve a string representing the floating-point variable V. Also, we use Status'Image (V) to retrieve a string representing the textual version of the Status.

In the Ada Reference Manual

- Image Attributes¹⁴¹

¹⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

Type'Image and Obj'Image

We can also apply the Image attribute to an object directly:

Listing 830: show_simple_image.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Image is
4   V : Integer;
5 begin
6   V := 10;
7   Put_Line ("V: " & V'Image);
8
9   -- Equivalent to:
10  -- Put_Line ("V: " & Integer'Image (V));
11 end Show_Simple_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Simple_Image
 MD5: c8b2e458de47b403568dd795b3d3fc24

Runtime output

V: 10

In this example, the **Integer'Image** (V) and V'Image forms are equivalent.

Wider versions of Image

Although we've been talking only about the Image attribute, it's important to mention that each of the wider versions of the string types also has a corresponding Image attribute. In fact, this is the attribute for each string type:

Attribute	Type of Returned String
Image	String
Wide_Image	Wide_String
Wide_Wide_Image	Wide_Wide_String

Let's see a simple example:

Listing 831: show_wide_wide_image.adb

```

1 with Ada.Wide_Wide_Text_IO;
2 use Ada.Wide_Wide_Text_IO;
3
4 procedure Show_Wide_Wide_Image is
5   F : Float;
6 begin
7   F := 100.0;
8   Put_Line ("F = "
9             & F'Wide_Wide_Image);
10 end Show_Wide_Wide_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Wide_Wide_Image
 MD5: ff542ef93286529343466c27935d5c21

Runtime output

```
F = 1.00000E+02
```

In this example, we use the `Wide_Wide_Image` attribute to retrieve a string of `Wide_Wide_String` type for the floating-point variable `F`.

Image attribute for non-scalar types

Note

This feature was introduced in Ada 2022.

In the previous code examples, we were using the `Image` attribute with scalar types, but it isn't restricted to those types. In fact, we can also use this attribute when dealing with non-scalar types. For example:

Listing 832: `simple_records.ads`

```
1 package Simple_Records is
2
3     type Rec is limited private;
4
5     type Rec_Access is access Rec;
6
7     function Init return Rec;
8
9     type Null_Rec is null record;
10
11 private
12
13     type Rec is limited record
14         F : Float;
15         I : Integer;
16     end record;
17
18     function Init return Rec is
19         ((F => 10.0, I => 4));
20
21 end Simple_Records;
```

Listing 833: `show_non_scalar_image.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Unchecked_Deallocation;
3
4 with Simple_Records;
5 use Simple_Records;
6
7 procedure Show_Non_Scalar_Image is
8
9     procedure Free is
10         new Ada.Unchecked_Deallocation
11             (Object => Rec,
12              Name   => Rec_Access);
13
14     R_A : Rec_Access :=
15         new Rec'(Init);
16
```

(continues on next page)

(continued from previous page)

```

17     N_R : Null_Rec :=
18         (null record);
19 begin
20     R_A := new Rec'(Init);
21     N_R := (null record);
22
23     Put_Line ("R_A:      " & R_A'Image);
24     Put_Line ("R_A.all: " & R_A.all'Image);
25     Put_Line ("N_R:      " & N_R'Image);
26
27     Free (R_A);
28     Put_Line ("R_A:      " & R_A'Image);
29 end Show_Non_Scalar_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Non_Scalar_Image
MD5: eb48f3fbe69b70258bc26f467918717c

Runtime output

```

R_A:      (access 5a3d114f12c0)
R_A.all:
(F =>  1.000000E+01,
 I =>  4)
N_R:      (NULL RECORD)
R_A:      null

```

In the `Show_Non_Scalar_Image` procedure from this example, we display the access value of `R_A` and the contents of the dereferenced access object (`R_A.all`). Also, we see the indication that `N_R` is a null record and `R_A` is null after the call to `Free`.

Historically

Since Ada 2022, the `Image` attribute is available for all types. Prior to this version of the language, it was only available for scalar types. (For other kind of types, programmers had to use the `Image` attribute for each component of a record, for example.)

In fact, prior to Ada 2022, the `Image` attribute was described in the [3.5 Scalar Types](http://www.ada-auth.org/standards/22rm/html/RM-3-5.html)¹⁴² section of the Ada Reference Manual, as it was only applied to those types. Now, it is part of the new [Image Attributes](http://www.ada-auth.org/standards/22rm/html/RM-4-10.html)¹⁴³ section.

Let's see another example, this time with arrays:

Listing 834: `show_array_image.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Array_Image is
4
5      type Float_Array is
6          array (Positive range <>) of Float;
7
8      FA_3C   : Float_Array (1 .. 3);
9      FA_Null : Float_Array (1 .. 0);
10
11 begin

```

(continues on next page)

¹⁴² <http://www.ada-auth.org/standards/22rm/html/RM-3-5.html>

¹⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

(continued from previous page)

```
12  FA_3C    := [1.0, 3.0, 2.0];
13  FA_Null := [];
14
15  Put_Line ("FA_3C:  " & FA_3C'Image);
16  Put_Line ("FA_Null: " & FA_Null'Image);
17  end Show_Array_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Array_Image
MD5: a24daba1d92a139ae8995bba5a81e0d6

Runtime output

```
FA_3C:
[ 1.00000E+00,  3.00000E+00,  2.00000E+00]
FA_Null:
[]
```

In this example, we display the values of the three components of the FA_3C array. Also, we display the null array FA_Null.

Image attribute for tagged types

In addition to untagged types, we can also use the Image attribute with tagged types. For example:

Listing 835: simple_records.ads

```
1  package Simple_Records is
2
3      type Rec is tagged limited private;
4
5      function Init return Rec;
6
7      type Rec_Child is new Rec with private;
8
9      overriding function Init return Rec_Child;
10
11  private
12
13      type Status is (Unknown, Off, On);
14
15      type Rec is tagged limited record
16          F : Float;
17          I : Integer;
18      end record;
19
20      function Init return Rec is
21          ((F => 10.0, I => 4));
22
23      type Rec_Child is new Rec with record
24          Z : Status;
25      end record;
26
27      function Init return Rec_Child is
28          (Rec'(Init) with Z => Off);
29
30  end Simple_Records;
```

Listing 836: show_tagged_image.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2
3  with Simple_Records; use Simple_Records;
4
5  procedure Show_Tagged_Image is
6      R      : constant Rec      := Init;
7      R_Class : constant Rec'Class := Rec'(Init);
8      R_C     : constant Rec_Child := Init;
9  begin
10     Put_Line ("R:      " & R'Image);
11     Put_Line ("R_Class: " & R_Class'Image);
12     Put_Line ("R_A:     " & R_C'Image);
13 end Show_Tagged_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Tagged_Image
MD5: 496827d5f81f8b7bec3b1d4a104f550e

Runtime output

```

R:
(F =>  1.00000E+01,
 I =>  4)
R_Class: SIMPLE_RECORDS.REC'
(F =>  1.00000E+01,
 I =>  4)
R_A:
(F =>  1.00000E+01,
 I =>  4,
 Z => OFF)

```

In the `Show_Tagged_Image` procedure from this example, we display the contents of the `R` object of `Rec` type and the `R_Class` object of `Rec'Class` type. Also, we display the contents of the `R_C` object of the `Rec_Child` type, which is derived from the `Rec` type.

Image attribute for task and protected types

We can also apply the `Image` attribute to protected objects and tasks:

Listing 837: simple_tasking.ads

```

1  package Simple_Tasking is
2
3      protected type Protected_Float (I : Integer) is
4
5      private
6          V : Float := Float (I);
7      end Protected_Float;
8
9      protected type Protected_Null is
10     private
11     end Protected_Null;
12
13     task type T is
14         entry Start;
15     end T;
16
17 end Simple_Tasking;

```

Listing 838: simple_tasking.adb

```
1 package body Simple_Tasking is
2
3     protected body Protected_Float is
4
5     end Protected_Float;
6
7     protected body Protected_Null is
8
9     end Protected_Null;
10
11    task body T is
12    begin
13        accept Start;
14    end T;
15
16 end Simple_Tasking;
```

Listing 839: show_protected_task_image.adb

```
1 with Ada.Text_IO;    use Ada.Text_IO;
2
3 with Simple_Tasking; use Simple_Tasking;
4
5 procedure Show_Protected_Task_Image is
6
7     PF : Protected_Float (0);
8     PN : Protected_Null;
9     T1 : T;
10
11 begin
12     Put_Line ("PF: " & PF'Image);
13     Put_Line ("PN: " & PN'Image);
14     Put_Line ("T1: " & T1'Image);
15
16     T1.Start;
17 end Show_Protected_Task_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Image_Attribute.Protected_Task_↵Image
MD5: feb14f17ba1cca0311420272ef91ab38

Runtime output

```
PF: (protected object)
PN: (protected object)
T1: (task t1_0000565A145A9090)
```

In this example, we display information about the protected object PF, the componentless protected object PN and the task T1.

25.7.6 Put_Image aspect

Note

This feature was introduced in Ada 2022.

Overview

In the previous section, we discussed many details about the Image attribute. In the code examples from that section, we've seen the default behavior of this attribute: the string returned by the calls to Image was always in the format defined by the Ada standard.

In some situations, however, we might want to customize the string that is returned by the Image attribute of a type T. Ada allows us to do that via the Put_Image aspect. This is what we have to do:

1. Specify the Put_Image aspect for the type T and indicate a procedure with a specific parameter profile — let's say, for example, a procedure named P.
2. Implement the procedure P and write the information we want to use into a buffer (by calling the routines defined for Root_Buffer_Type, such as the Put procedure).

We can see these steps performed in the code example below:

Listing 840: show_put_image.ads

```

1 with Ada.Strings.Text_Buffers;
2
3 package Show_Put_Image is
4
5     type T is null record
6         with Put_Image => Put_Image_T;
7         -- ^ Custom version of Put_Image
8
9     use Ada.Strings.Text_Buffers;
10
11     procedure Put_Image_T
12         (Buffer : in out Root_Buffer_Type'Class;
13          Arg     : T);
14
15 end Show_Put_Image;
```

Listing 841: show_put_image.adb

```

1 package body Show_Put_Image is
2
3     procedure Put_Image_T
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg     : T) is
6         pragma Unreferenced (Arg);
7     begin
8         -- Call Put with customized
9         -- information
10        Buffer.Put ("<custom info>");
11    end Put_Image_T;
12
13 end Show_Put_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Simple_Put_Image
MD5: 45c55444f0e1825312b5eafe307ca58d

In the Show_Put_Image package, we use the Put_Image aspect in the declaration of the T type. There, we indicate that the Image attribute shall use the Put_Image_T procedure instead of the default version.

In the body of the Put_Image_T procedure, we implement our custom version of the Image attribute. We do that by calling the Put procedure with the information we want to provide in the Image attribute. Here, we access a buffer of Root_Buffer_Type type, which is defined

in the `Ada.Strings.Text_Buffers` package. (We discuss more about this package *later on* (page 619).)

In the Ada Reference Manual

- [Image Attributes](#)¹⁴⁴

Complete Example of Put_Image

Let's see a complete example in which we use the `Put_Image` aspect and write useful information to the buffer:

Listing 842: `custom_numerics.ads`

```
1 with Ada.Strings.Text_Buffers;
2
3 package Custom_Numerics is
4
5     type Float_Integer is record
6         F : Float      := 0.0;
7         I : Integer     := 0;
8     end record
9     with Dynamic_Predicate =>
10         Integer (Float_Integer.F) =
11             Float_Integer.I,
12         Put_Image      => Put_Float_Integer;
13     -- ^ Custom version of Put_Image
14
15     use Ada.Strings.Text_Buffers;
16
17     procedure Put_Float_Integer
18         (Buffer : in out Root_Buffer_Type'Class;
19          Arg     : Float_Integer);
20
21 end Custom_Numerics;
```

Listing 843: `custom_numerics.adb`

```
1 package body Custom_Numerics is
2
3     procedure Put_Float_Integer
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg     : Float_Integer) is
6     begin
7         -- Call Wide_Wide_Put with customized
8         -- information
9         Buffer.Wide_Wide_Put
10             ("(F : " & Arg.F'Wide_Wide_Image & ", "
11              & "I : " & Arg.I'Wide_Wide_Image & ")");
12     end Put_Float_Integer;
13
14 end Custom_Numerics;
```

Listing 844: `show_put_image.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Custom_Numerics; use Custom_Numerics;
```

(continues on next page)

¹⁴⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-10.html>

(continued from previous page)

```

4
5 procedure Show_Put_Image is
6   V : Float_Integer;
7 begin
8   V := (F => 100.2,
9         I => 100);
10  Put_Line ("V = "
11           & V'Image);
12 end Show_Put_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Put_Image_Custom_↵Numerics

MD5: 1dbb5fa612b5ca86facc3e93b47977e0

Runtime output

```
V = (F : 1.00200E+02, I : 100)
```

In the Custom_Numerics package of this example, we specify the Put_Image aspect and indicate the Put_Float_Integer procedure. In that procedure, we display the information of components F and I. Then, in the Show_Put_Image procedure, we use the Image attribute for the V variable and see the information in the exact format we specified. (If you like to see the default version of the Put_Image instead, you may comment out the Put_Image aspect part in the declaration of Float_Integer.)

Relation to the Image attribute

Note that we cannot override the Image attribute directly — there's no Image *aspect* that we could specify. However, as we've just seen, we can do this indirectly by using our own version of the Put_Image procedure for a type T.

The Image attribute of a type T makes use of the procedure indicated in the Put_Image aspect. Let's say we have the following declaration:

```

type T is null record
  with Put_Image => Put_Image_T;
```

When we then use the T'Image attribute in our code, the custom Put_Image_T procedure is automatically called. This is a simplified example of how the Image function is implemented:

```

function Image (V : T)
  return String is
  Buffer : Custom_Buffer;
  --      ^ of Root_Buffer_Type'Class
begin
  -- Calling Put_Image procedure
  -- for type T
  Put_Image_T (Buffer, V);

  -- Retrieving the text from the
  -- buffer as a string
  return Buffer.Get;
end Image;
```

In other words, the Image attribute basically:

- calls the Put_Image procedure specified in the Put_Image aspect of type T's declaration and passes a buffer;

and

- retrieves the contents of the buffer as a string and returns it.

If the `Put_Image` aspect of type `T` isn't specified, the default version is used. (We've seen the default version of various types *in the previous section* (page 605) about the `Image` attribute.)

Put_Image and derived types

Types that were derived from untagged types (or null extensions) make use of the `Put_Image` procedure that was specified for their parent type — either a custom procedure indicated in the `Put_Image` aspect or the default one. Naturally, if a derived type has the `Put_Image` aspect, the procedure indicated in the aspect is used instead. For example:

Listing 845: `untagged_put_image.ads`

```
1 with Ada.Strings.Text_Buffers;
2
3 package Untagged_Put_Image is
4
5     use Ada.Strings.Text_Buffers;
6
7     type T is null record
8         with Put_Image => Put_Image_T;
9
10    procedure Put_Image_T
11        (Buffer : in out Root_Buffer_Type'Class;
12         Arg    : T);
13
14    type T_Derived_1 is new T;
15
16    type T_Derived_2 is new T
17        with Put_Image => Put_Image_T_Derived_2;
18
19    procedure Put_Image_T_Derived_2
20        (Buffer : in out Root_Buffer_Type'Class;
21         Arg    : T_Derived_2);
22
23 end Untagged_Put_Image;
```

Listing 846: `untagged_put_image.adb`

```
1 package body Untagged_Put_Image is
2
3     procedure Put_Image_T
4         (Buffer : in out Root_Buffer_Type'Class;
5          Arg    : T) is
6         pragma Unreferenced (Arg);
7     begin
8         Buffer.Wide_Wide_Put ("Put_Image_T");
9     end Put_Image_T;
10
11    procedure Put_Image_T_Derived_2
12        (Buffer : in out Root_Buffer_Type'Class;
13         Arg    : T_Derived_2) is
14        pragma Unreferenced (Arg);
15    begin
16        Buffer.Wide_Wide_Put
17            ("Put_Image_T_Derived_2");
18    end Put_Image_T_Derived_2;
19
```

(continues on next page)

(continued from previous page)

```
20 end Untagged_Put_Image;
```

Listing 847: show_untagged_put_image.adb

```
1 with Ada.Text_IO;          use Ada.Text_IO;
2
3 with Untagged_Put_Image; use Untagged_Put_Image;
4
5 procedure Show_Untagged_Put_Image is
6   Obj_T          : T;
7   Obj_T_Derived_1 : T_Derived_1;
8   Obj_T_Derived_2 : T_Derived_2;
9 begin
10  Put_Line ("T'Image : "
11           & Obj_T'Image);
12  Put_Line ("T_Derived_1'Image : "
13           & Obj_T_Derived_1'Image);
14  Put_Line ("T_Derived_2'Image : "
15           & Obj_T_Derived_2'Image);
16 end Show_Untagged_Put_Image;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Untagged_Put_Image
MD5: acc0c17d45e6271cb582e65bfc8a2a98

Runtime output

```
T'Image :          Put_Image_T
T_Derived_1'Image : Put_Image_T
T_Derived_2'Image : Put_Image_T_Derived_2
```

In this example, we declare the type `T` and its derived types `T_Derived_1` and `T_Derived_2`. When running this code, we see that:

- `T_Derived_1` makes use of the `Put_Image_T` procedure from its parent.
 - Note that, if we remove the `Put_Image` aspect from the declaration of `T`, the default version of the `Put_Image` procedure is used for both `T` and `T_Derived_1` types.
- `T_Derived_2` makes use of the `Put_Image_T_Derived_2` procedure, which was indicated in the `Put_Image` aspect of that type, instead of its parent's procedure.

Put_Image and tagged types

Types that are derived from a tagged type may also inherit the `Put_Image` aspect. However, there are a couple of small differences in comparison to untagged types, as we can see in the following example:

Listing 848: tagged_put_image.ads

```
1 with Ada.Strings.Text_Buffers;
2
3 package Tagged_Put_Image is
4
5   use Ada.Strings.Text_Buffers;
6
7   type T is tagged record
8     I : Integer := 0;
9   end record
```

(continues on next page)

(continued from previous page)

```

10     with Put_Image => Put_Image_T;
11
12     procedure Put_Image_T
13       (Buffer : in out Root_Buffer_Type'Class;
14        Arg     : T);
15
16     type T_Child_1 is new T with record
17       I1 : Integer;
18     end record;
19
20     type T_Child_2 is new T with null record;
21
22     type T_Child_3 is new T with record
23       I3 : Integer := 0;
24     end record
25     with Put_Image => Put_Image_T_Child_3;
26
27     procedure Put_Image_T_Child_3
28       (Buffer : in out Root_Buffer_Type'Class;
29        Arg     : T_Child_3);
30
31 end Tagged_Put_Image;

```

Listing 849: tagged_put_image.adb

```

1  package body Tagged_Put_Image is
2
3     procedure Put_Image_T
4       (Buffer : in out Root_Buffer_Type'Class;
5        Arg     : T) is
6       pragma Unreferenced (Arg);
7     begin
8       Buffer.Wide_Wide_Put ("Put_Image_T");
9     end Put_Image_T;
10
11    procedure Put_Image_T_Child_3
12      (Buffer : in out Root_Buffer_Type'Class;
13       Arg     : T_Child_3) is
14      pragma Unreferenced (Arg);
15    begin
16      Buffer.Wide_Wide_Put
17        ("Put_Image_T_Child_3");
18    end Put_Image_T_Child_3;
19
20 end Tagged_Put_Image;

```

Listing 850: show_tagged_put_image.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Tagged_Put_Image; use Tagged_Put_Image;
4
5  procedure Show_Tagged_Put_Image is
6    Obj_T           : T;
7    Obj_T_Child_1   : T_Child_1;
8    Obj_T_Child_2   : T_Child_2;
9    Obj_T_Child_3   : T_Child_3;
10  begin
11    Put_Line ("T'Image :      "
12             & Obj_T'Image);

```

(continues on next page)

(continued from previous page)

```

13   Put_Line ("-----");
14   Put_Line ("T_Child_1'Image : "
15           & Obj_T_Child_1'Image);
16   Put_Line ("-----");
17   Put_Line ("T_Child_2'Image : "
18           & Obj_T_Child_2'Image);
19   Put_Line ("-----");
20   Put_Line ("T_Child_3'Image : "
21           & Obj_T_Child_3'Image);
22   Put_Line ("-----");
23   Put_Line ("T'Class'Image : "
24           & T'Class (Obj_T_Child_1)'Image);
25 end Show_Tagged_Put_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Put_Image.Tagged_Put_Image
MD5: b19214bbcb8c0339ead744afffcdd68

Runtime output

```

T'Image :          Put_Image_T
-----
T_Child_1'Image :
(Put_Image_T with I1 => 0)
-----
T_Child_2'Image : Put_Image_T
-----
T_Child_3'Image : Put_Image_T_Child_3
-----
T'Class'Image :   TAGGED_PUT_IMAGE.T_CHILD_1'
(Put_Image_T with I1 => 0)

```

In this example, we declare the type `T` and its derived types `T_Child_1`, `T_Child_2` and `T_Child_3`. When running this code, we see that:

- for both `T_Child_1` and `T_Child_2`, the parent's `Put_Image` aspect (the `Put_Image_T` procedure) is called and its information is combined with the information from the type extension;
 - For the `T_Child_1` type, the `I1` component of the type extension is displayed by calling a default version of the `Put_Image` procedure for that component — (`Put_Image_T` with `I1 => 0`) is displayed.
 - For the `T_Child_2` type, no additional information is displayed because this type has a null extension.
- for the `T_Child_3` type, the `Put_Image_T_Child_3` procedure, which was indicated in the `Put_Image` aspect of the type, is used.

Finally, class-wide types (such as `T'Class`) include additional information. Here, the tag of the specific derived type is displayed first — in this case, the tag of the `T_Child_1` type — and then the actual information for the derived type is displayed.

25.7.7 Universal text buffer

In the [previous section](#) (page 612), we've seen that the first parameter of the procedure indicated in the `Put_Image` aspect has the `Root_Buffer_Type'Class` type, which is defined in the `Ada.Strings.Text_Buffers` package. In this section, we talk more about this type and additional procedures associated with this type.

Note

This feature was introduced in Ada 2022.

Overview

We use the `Root_Buffer_Type'Class` type to implement a universal text buffer that is used to store and retrieve information about data types. Because this text buffer isn't associated with specific data types, it is universal — in the sense that we can really use it for any data type, regardless of the characteristics of this type.

In theory, we could use Ada's universal text buffer to implement applications that actually process text in some form — for example, when implementing a text editor. However, in general, Ada programmers are only expected to make use of the `Root_Buffer_Type'Class` type when implementing a procedure for the `Put_Image` aspect. For this reason, we won't discuss any kind of type derivation — or any other kind of usages of this type — in this section. Instead, we'll just focus on additional subprograms from the `Ada.Strings.Text_Buffers` package.

In the Ada Reference Manual

- [Universal Text Buffers](#)¹⁴⁵

Additional procedures

In the previous section, we used the `Put` procedure — and the related `Wide_Put` and `Wide_Wide_Put` procedures — from the `Ada.Strings.Text_Buffers` package. In addition to these procedures, the package also includes:

- the `New_Line` procedure, which writes a new line marker to the text buffer;
- the `Increase_Indent` procedure, which increases the indentation in the text buffer; and
- the `Decrease_Indent` procedure, which decreases the indentation in the text buffer.

The `Ada.Strings.Text_Buffers` package also includes the `Current_Indent` function, which retrieves the current indentation counter.

Let's revisit an example from the previous section and use the procedures mentioned above:

Listing 851: `custom_numerics.ads`

```
1 with Ada.Strings.Text_Buffers;
2
3 package Custom_Numerics is
4
5     type Float_Integer is record
6         F : Float;
7         I : Integer;
8     end record
9     with Dynamic_Predicate =>
10         Integer (Float_Integer.F) =
11             Float_Integer.I,
12         Put_Image          => Put_Float_Integer;
13     -- ^ Custom version of Put_Image
14
15 use Ada.Strings.Text_Buffers;
```

(continues on next page)

¹⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-4-12.html>

(continued from previous page)

```

16
17   procedure Put_Float_Integer
18   (Buffer : in out Root_Buffer_Type'Class;
19    Arg    :      Float_Integer);
20
21 end Custom_Numerics;

```

Listing 852: custom_numerics.adb

```

1  package body Custom_Numerics is
2
3      procedure Put_Float_Integer
4      (Buffer : in out Root_Buffer_Type'Class;
5       Arg    :      Float_Integer) is
6      begin
7          Buffer.Wide_Wide_Put "(");
8          Buffer.New_Line;
9
10         Buffer.Increase_Indent;
11
12         Buffer.Wide_Wide_Put
13         ("F : "
14          & Arg.F'Wide_Wide_Image);
15         Buffer.New_Line;
16
17         Buffer.Wide_Wide_Put
18         ("I : "
19          & Arg.I'Wide_Wide_Image);
20
21         Buffer.Decrease_Indent;
22         Buffer.New_Line;
23
24         Buffer.Wide_Wide_Put (")");
25     end Put_Float_Integer;
26
27 end Custom_Numerics;

```

Listing 853: show_put_image.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Custom_Numerics; use Custom_Numerics;
4
5  procedure Show_Put_Image is
6      V : Float_Integer;
7  begin
8      V := (F => 100.2,
9            I => 100);
10     Put_Line ("V = "
11              & V'Image);
12 end Show_Put_Image;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Strings.Universal_Text_Buffer.Put_Image_
↳ Custom_Numerics
MD5: e976a2ade2ad4a10033924e19bc84159

Runtime output


```
V = (  
  F : 1.00200E+02  
  I : 100  
)
```

In the body of the `Put_Float_Integer` procedure, we're using the `New_Line`, `Increase_Indent` and `Decrease_Indent` procedures to improve the format of the string returned by the `Float_Integer`' `Image` attribute. Using these procedures, you can create any kind of output format for your custom type.

25.8 Numerics

25.8.1 Numeric Literals

Classification

We've already discussed basic characteristics of numeric literals in the Introduction to Ada course — although we haven't used this terminology there. There are two kinds of numeric literals in Ada: integer literals and real literals. They are distinguished by the absence or presence of a radix point. For example:

Listing 854: `real_integer_literals.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Real_Integer_Literals is  
4   Integer_Literal : constant := 365;  
5   Real_Literal    : constant := 365.2564;  
6 begin  
7   Put_Line ("Integer Literal: "  
8             & Integer_Literal'Image);  
9   Put_Line ("Real Literal: "  
10            & Real_Literal'Image);  
11 end Real_Integer_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Real_Integer_Literals
MD5: balcc348cad054f3ab86c05e051b40fa

Runtime output

```
Integer Literal: 365  
Real Literal:   3.652564000000000000E+02
```

In this example, `365` is an integer literal and `365.2564` is a real literal.

Another classification takes the use of a base indicator into account. (Remember that, when writing a literal such as `2#1011#`, the base is the element before the first `#` sign.) So here we distinguish between decimal literals and based literals. For example:

Listing 855: `decimal_based_literals.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Decimal_Based_Literals is  
4  
5   package F_IO is new  
6     Ada.Text_IO.Float_IO (Float);
```

(continues on next page)

(continued from previous page)

```

7
8  --
9  --  DECIMAL LITERALS
10 --
11
12 Dec_Integer  : constant := 365;
13
14 Dec_Real     : constant := 365.2564;
15 Dec_Real_Exp : constant := 0.365_256_4e3;
16
17 --
18 --  BASED LITERALS
19 --
20
21 Based_Integer  : constant := 16#16D#;
22 Based_Integer_Exp : constant := 5#243#e1;
23
24 Based_Real     : constant :=
25 2#1_0110_1101.0100_0001_1010_0011_0111#;
26 Based_Real_Exp : constant :=
27 7#1.031_153_643#e3;
28 begin
29   F_IO.Default_Fore := 3;
30   F_IO.Default_Aft  := 4;
31   F_IO.Default_Exp  := 0;
32
33   Put_Line ("Dec_Integer:      "
34             & Dec_Integer'Image);
35
36   Put ("Dec_Real:              ");
37   F_IO.Put (Item => Dec_Real);
38   New_Line;
39
40   Put ("Dec_Real_Exp:          ");
41   F_IO.Put (Item => Dec_Real_Exp);
42   New_Line;
43
44   Put_Line ("Based_Integer:    "
45             & Based_Integer'Image);
46   Put_Line ("Based_Integer_Exp: "
47             & Based_Integer_Exp'Image);
48
49   Put ("Based_Real:            ");
50   F_IO.Put (Item => Based_Real);
51   New_Line;
52
53   Put ("Based_Real_Exp:        ");
54   F_IO.Put (Item => Based_Real_Exp);
55   New_Line;
56 end Decimal_Based_Literals;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Decimal_Based_Literals
MD5: bde8f422c3844826819348d18fb48a33

Runtime output

```

Dec_Integer:      365
Dec_Real:         365.2564

```

(continues on next page)

(continued from previous page)

```
Dec_Real_Exp:    365.2564
Based_Integer:   365
Based_Integer_Exp: 365
Based_Real:      365.2564
Based_Real_Exp:  365.2564
```

Based literals use the `base#number#` format. Also, they aren't limited to simple integer literals such as `16#16D#`. In fact, we can use a radix point or an exponent in based literals, as well as underscores. In addition, we can use any base from 2 up to 16. We discuss these aspects further in the next section.

Features and Flexibility

Note

This section was originally written by Franco Gasperoni and published as [Gem #7: The Beauty of Numeric Literals in Ada](#)¹⁴⁶.

Ada provides a simple and elegant way of expressing numeric literals. One of those simple, yet powerful aspects is the ability to use underscores to separate groups of digits. For example, `3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510` is more readable and less error prone to type than `3.14159265358979323846264338327950288419716939937510`. Here's the complete code:

Listing 856: `ada_numeric_literals.adb`

```
1 with Ada.Text_IO;
2
3 procedure Ada_Numeric_Literals is
4   Pi    : constant :=
5     3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510;
6
7   Pi2   : constant :=
8     3.14159265358979323846264338327950288419716939937510;
9
10  Z      : constant := Pi - Pi2;
11  pragma Assert (Z = 0.0);
12
13  use Ada.Text_IO;
14  begin
15    Put_Line ("Z = " & Float'Image (Z));
16  end Ada_Numeric_Literals;
```

Code block metadata

Project: `Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Pi_Literals`
 MD5: `8f6516730fa98f08234b159488431aaf`

Runtime output

```
Z = 0.00000E+00
```

Also, when using based literals, Ada allows any base from 2 to 16. Thus, we can write the decimal number 136 in any one of the following notations:

¹⁴⁶ <https://www.adacore.com/gems/ada-gem-7>

Listing 857: ada_numeric_literals.adb

```

1  with Ada.Text_IO;
2
3  procedure Ada_Numeric_Literals is
4      Bin_136 : constant := 2#1000_1000#;
5      Oct_136 : constant := 8#210#;
6      Dec_136 : constant := 10#136#;
7      Hex_136 : constant := 16#88#;
8      pragma Assert (Bin_136 = 136);
9      pragma Assert (Oct_136 = 136);
10     pragma Assert (Dec_136 = 136);
11     pragma Assert (Hex_136 = 136);
12
13     use Ada.Text_IO;
14
15     begin
16         Put_Line ("Bin_136 = "
17             & Integer'Image (Bin_136));
18         Put_Line ("Oct_136 = "
19             & Integer'Image (Oct_136));
20         Put_Line ("Dec_136 = "
21             & Integer'Image (Dec_136));
22         Put_Line ("Hex_136 = "
23             & Integer'Image (Hex_136));
24     end Ada_Numeric_Literals;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Based_Literals
MD5: 0959ec5e4aafcde245c5a15597ac9b7e

Runtime output

```

Bin_136 = 136
Oct_136 = 136
Dec_136 = 136
Hex_136 = 136

```

** In other languages**

The rationale behind the method to specify based literals in the C programming language is strange and unintuitive. Here, you have only three possible bases: 8, 10, and 16 (why no base 2?). Furthermore, requiring that numbers in base 8 be preceded by a zero feels like a bad joke on us programmers. For example, what values do **0210** and **210** represent in C?

When dealing with microcontrollers, we might encounter I/O devices that are memory mapped. Here, we have the ability to write:

```

Lights_On   : constant := 2#1000_1000#;
Lights_Off  : constant := 2#0111_0111#;

```

and have the ability to turn on/off the lights as follows:

```

Output_Devices := Output_Devices or Lights_On;
Output_Devices := Output_Devices and Lights_Off;

```

Here's the complete example:

Listing 858: ada_numeric_literals.adb

```

1  with Ada.Text_IO;
2
3  procedure Ada_Numeric_Literals is
4      Lights_On   : constant := 2#1000_1000#;
5      Lights_Off  : constant := 2#0111_0111#;
6
7      type Byte is mod 256;
8      Output_Devices : Byte := 0;
9
10     -- for Output_Devices'Address
11     -- use 16#DEAD_BEEF#;
12     -- ~~~~~
13     -- Memory mapped Output
14
15     use Ada.Text_IO;
16 begin
17     Output_Devices := Output_Devices or
18                       Lights_On;
19
20     Put_Line ("Output_Devices (lights on ) = "
21              & Byte'Image (Output_Devices));
22
23     Output_Devices := Output_Devices and
24                       Lights_Off;
25
26     Put_Line ("Output_Devices (lights off) = "
27              & Byte'Image (Output_Devices));
28 end Ada_Numeric_Literals;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Lights
MD5: c3e72b25366d8d815a1f425f2695ad0b

Runtime output

```

Output_Devices (lights on ) = 136
Output_Devices (lights off) = 0

```

Of course, we can also use *records with representation clauses* (page 373) to do the above, which is even more elegant.

The notion of base in Ada allows for exponents, which is particularly pleasant. For instance, we can write:

Listing 859: literal_binaries.ads

```

1  package Literal_Binaries is
2      Kilobyte   : constant := 2#1#e+10;
3      Megabyte   : constant := 2#1#e+20;
4      Gigabyte   : constant := 2#1#e+30;
5      Terabyte   : constant := 2#1#e+40;
6      Petabyte   : constant := 2#1#e+50;
7      Exabyte    : constant := 2#1#e+60;
8      Zettabyte  : constant := 2#1#e+70;
9      Yottabyte  : constant := 2#1#e+80;
10 end Literal_Binaries;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Binary
MD5: 98d971e0f170db570069f8868e442d6d

In based literals, the exponent — like the base — uses the regular decimal notation and specifies the power of the base that the based literal should be multiplied with to obtain the final value. For instance `2#1#e+10` = 1×2^{10} = `1_024` (in base 10), whereas `16#F#e+2` = 15×16^2 = 15×256 = `3_840` (in base 10).

Based numbers apply equally well to real literals. We can, for instance, write:

```
One_Third : constant := 3#0.1#;  
--  
--           same as 1.0/3
```

Whether we write `3#0.1#` or `1.0 / 3`, or even `3#1.0#e-1`, Ada allows us to specify exactly rational numbers for which decimal literals cannot be written.

The last nice feature is that Ada has an open-ended set of integer and real types. As a result, numeric literals in Ada do not carry with them their type as, for example, in C. The actual type of the literal is determined from the context. This is particularly helpful in avoiding overflows, underflows, and loss of precision.

In other languages

In C, a source of confusion can be the distinction between `32l` and `32i`. Although both look similar, they're actually very different from each other.

And this is not all: all constant computations done at compile time are done in infinite precision, be they integer or real. This allows us to write constants with whatever size and precision without having to worry about overflow or underflow. We can for instance write:

```
Zero : constant := 1.0 - 3.0 * One Third;
```

and be guaranteed that constant Zero has indeed value zero. This is very different from writing:

```
One_Third_Approx : constant :=  
    0.3333333333333333333333333333;  
Zero_Approx      : constant :=  
    1.0 - 3.0 * One_Third_Approx;
```

where `Zero_Approx` is really `1.0e-29` — and that will show up in your numerical computations. The above is quite handy when we want to write fractions without any loss of precision. Here's the complete code:

Listing 860: ada_numeric_literals.adb

```
1 with Ada.Text_IO;
2
3 procedure Ada_Numeric_Literals is
4     One_Third : constant := 3#1.0#e-1;
5     -- same as 1.0/3.0
6
7     Zero      : constant := 1.0 - 3.0 * One_Third;
8     pragma Assert (Zero = 0.0);
9
10    One_Third_Approx : constant :=
11        0.33333333333333333333333333333333;
12    Zero_Approx      : constant :=
13        1.0 - 3.0 * One Third Approx;
```

(continues on next page)

(continued from previous page)

```
14
15     use Ada.Text_IO;
16
17 begin
18     Put_Line ("Zero           = "
19             & Float'Image (Zero));
20     Put_Line ("Zero_Approx = "
21             & Float'Image (Zero_Approx));
22 end Ada_Numeric_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literals
MD5: ee604245b34e8cb878a8ebdb21cd564e

Runtime output

```
Zero           = 0.00000E+00
Zero_Approx = 1.00000E-29
```

Along these same lines, we can write:

Listing 861: ada_numeric_literals.adb

```
1  with Ada.Text_IO;
2
3  with Literal_Binaries; use Literal_Binaries;
4
5  procedure Ada_Numeric_Literals is
6
7      Big_Sum : constant := 1          +
8                      Kilobyte  +
9                      Megabyte  +
10                     Gigabyte  +
11                     Terabyte  +
12                     Petabyte  +
13                     Exabyte   +
14                     Zettabyte;
15
16     Result : constant := (Yottabyte - 1) /
17                         (Kilobyte - 1);
18
19     Nil    : constant := Result - Big_Sum;
20     pragma Assert (Nil = 0);
21
22     use Ada.Text_IO;
23
24 begin
25     Put_Line ("Nil           = "
26             & Integer'Image (Nil));
27 end Ada_Numeric_Literals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Numeric_Literals.Literal_Binary
MD5: 7bda6442e68271d12bdb827b63f0d702

Runtime output

```
Nil           = 0
```

and be guaranteed that Nil is equal to zero.

25.8.2 Universal Numeric Types

Previously, we introduced the concept of *universal types* (page 302). Three of them are numeric types: universal real, universal integer and universal fixed types. In this section, we discuss these universal numeric types in more detail.

Universal Real and Integer

Universal real and integer types are mainly used in the declaration of *named numbers* (page 287):

Listing 862: show_universal_real_integer.ads

```

1 package Show_Universal_Real_Integer is
2
3   Pi : constant := 3.1415926535;
4       ^^^^^^^^^^^^^
5       universal real type
6
7   N  : constant := 10;
8       ^^
9       universal integer type
10
11 end Show_Universal_Real_Integer;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
 ↳ Universal_Real_Integer
 MD5: 3cfa52af66185c693ede07f3b0e689e6

The type of a named number is implied by the type of the *numeric literal* (page 622) and the type of any named numbers that we use in the *static expression* (page 629). (We discuss static expressions next.) In this specific example, we declare `Pi` using a real literal, which implies that it's a named number of universal real type. Likewise, `N` is of universal integer type because we use an integer literal in its declaration.

In the Ada Reference Manual

- [3.3.2 Number Declarations](#)¹⁴⁷

Static expressions

As we've just seen, we can use an expression in the declaration of a named number. This expression is static, as it's always evaluated at compile time. Therefore, we must use the keyword **constant** in the declaration of named numbers.

If all components of the static expression are of universal integer type, then the named number is of universal integer type. Otherwise, the static expression is of universal real type. For example, if the first element of a static expression is of universal integer type, but we have a constant of universal real type in the same expression, then the type of the whole static expression is universal real:

Listing 863: static_expressions.ads

```

1 package Static_Expressions is
2
```

(continues on next page)

¹⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-3-2.html>

(continued from previous page)

```

3  Two_Pi : constant := 2 * 3.1415926535;
4      ^
5      universal integer type
6
7      3.1415926535
8      ^^^^^^^^^^^
9      universal real type
10
11  => result: universal real type
12
13  end Static_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_Expressions
 MD5: 449286ea630771c9b673263af942bef6

In this example, the static expression is of universal real type because of the real literal (3.1415926535) — even though we have the universal integer 2 in the expression.

Likewise, if we use a constant of universal real type in the static expression, the result is of universal real type:

Listing 864: static_expressions.ads

```

1  package Static_Expressions is
2
3  Pi      : constant := 3.1415926535;
4      ^^^^^^^^^^^
5      universal real type
6
7  Two_Pi : constant := 2 * Pi;
8      ^
9      universal integer type
10
11      Pi
12      ^^
13      universal real type
14
15  => result: universal real type
16
17  end Static_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_Expressions
 MD5: 3de80a66631da876d877e7e3bbe73277

In this example, the result of the static expression is of universal real type because of we're using the named number Pi, which is of universal real type.

Complexity of static expressions

The operations that we use in static expressions may be arbitrarily complex. For example:

Listing 865: static_expressions.ads

```

1 package Static_Expressions is
2
3   C1 : constant := 300_453.5;
4   C2 : constant := 455_233.5 * C1;
5   C3 : constant := 872_922.5 * C2;
6   C4 : constant := 155_277.5 * C1 + C2 / C3;
7   C5 : constant := 2.0 * C1 +
8               3.0 * (C2 / (C4 * C3)) +
9               4.0 * (C1 / (C2 * C2)) +
10              5.0 * (C3 / (C1 * C1));
11
12 end Static_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_Expressions
 MD5: ebdd5b1c64ad1944931a962756e72291

As we can see in this example, we may create a chain of dependencies, where the result of a static expression depends on the result of previously evaluated static expressions. For instance, C5 depends on the evaluation of C1, C2, C3, C4.

Accuracy of static expressions

The accuracy and range of numeric literals used in static expressions may be arbitrarily high as well:

Listing 866: static_expressions.ads

```

1 package Static_Expressions is
2
3   Pi : constant :=
4       3.14159_26535_89793_23846_26433_83279_50288;
5
6   Seed : constant :=
7       143_574_786_272_784_656_928_283_872_972_764;
8
9   Super_Seed : constant :=
10      Seed * Seed * Seed * Seed * Seed * Seed;
11
12 end Static_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.Static_Expressions
 MD5: 777574a29ffa6da8bffb4287dee45be8

In this example, Super_Seed has a value that is above the typical range of integer constants. This might become challenging when using such named numbers in actual computations, as we *discuss soon* (page 635).

Another example is when the result of the expression is a *repeating decimal*¹⁴⁸:

¹⁴⁸ https://en.wikipedia.org/wiki/Repeating_decimal

Listing 867: repeating_decimals.ads

```

1 package Repeating_Decimals is
2
3   One_Over_Three : constant :=
4     1.0 / 3.0;
5
6 end Repeating_Decimals;
```

Listing 868: show_repeating_decimals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Repeating_Decimals;
4 use Repeating_Decimals;
5
6 procedure Show_Repeating_Decimals is
7   F_1_3 : constant Float :=
8     One_Over_Three;
9   LF_1_3 : constant Long_Float :=
10     One_Over_Three;
11   LLF_1_3 : constant Long_Long_Float :=
12     One_Over_Three;
13 begin
14   Put_Line (F_1_3'Image);
15   Put_Line (LF_1_3'Image);
16   Put_Line (LLF_1_3'Image);
17 end Show_Repeating_Decimals;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
 ↳ Repeating_Decimal
 MD5: 4fc38ef6482e403d655b4662d4199abb

Runtime output

```

3.33333E-01
3.333333333333333E-01
3.3333333333333333E-01
```

In this example, as expected, we see that the accuracy of the value we display increases if we use a type with higher precision. This wouldn't be possible if we had used a floating-point type with limited precision for the `One_Over_Three` constant:

Listing 869: repeating_decimals.ads

```

1 package Repeating_Decimals is
2
3   One_Over_Three : constant Long_Float :=
4     1.0 / 3.0;
5   --
6   --           ~~~~~~
7   --           using Long_Float instead of
8   --           universal real type
9 end Repeating_Decimals;
```

Listing 870: show_repeating_decimals.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 with Repeating_Decimals;
4 use Repeating_Decimals;
5
6 procedure Show_Repeating_Decimals is
7   F_1_3 : constant Float :=
8           Float (One_Over_Three);
9   LF_1_3 : constant Long_Float :=
10           Long_Float (One_Over_Three);
11   LLF_1_3 : constant Long_Long_Float :=
12           Long_Long_Float (One_Over_Three);
13 begin
14   Put_Line (F_1_3'Image);
15   Put_Line (LF_1_3'Image);
16   Put_Line (LLF_1_3'Image);
17 end Show_Repeating_Decimals;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
 ↳ Repeating_Decimal
 MD5: d0fa105d679cc246e2e8baf37cbe48c4

Runtime output

```

3.33333E-01
3.333333333333333E-01
3.33333333333333315E-01

```

Because we're using the **Long_Float** type for the `One_Over_Three` constant instead of the universal real type, the accuracy doesn't increase when we use the **Long_Long_Float** type — as we see in the value of the `LLF_1_3` constant — even though this type has a higher precision.

For further reading...

When using *big numbers* (page 683), you could simply assign the named number `One_Over_Three` to a big real:

Listing 871: repeating_decimals.ads

```

1 package Repeating_Decimals is
2
3   One_Over_Three : constant :=
4     1.0 / 3.0;
5
6 end Repeating_Decimals;

```


Conversion of universal real and integer

Although a named number exists as an numeric representation form in Ada, the value it represents cannot be used directly at runtime — even if we *just* display the value of the constant at runtime, for example. In fact, a conversion to a non-universal type is required in order to use the named number anywhere else other than a static expression:

Listing 874: static_expressions.ads

```

1 package Static_Expressions is
2
3   Pi : constant :=
4     3.14159_26535_89793_23846_26433_83279_50288;
5
6   Seed : constant :=
7     143_574_786_272_784_656_928_283_872_972_764;
8
9   Super_Seed : constant :=
10     Seed * Seed * Seed * Seed * Seed * Seed;
11
12 end Static_Expressions;
```

Listing 875: show_static_expressions.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Static_Expressions;
4 use Static_Expressions;
5
6 procedure Show_Static_Expressions is
7 begin
8   Put_Line (Pi'Image);
9   -- Same as:
10  -- Put_Line (Float (Pi)'Image);
11
12  Put_Line (Seed'Image);
13  -- Same as:
14  -- Put_Line (
15  --   Long_Long_Long_Integer (Seed)'Image);
16 end Show_Static_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
 ↪ Conversion_To_Non_Universal_Types
 MD5: e50641737f970b935e853ac249dd83d8

Runtime output

```

3.14159265358979324E+00
143574786272784656928283872972764
```

As we see in this example, the named number `Pi` is converted to **Float** before being used as an actual parameter in the call to `Put_Line`. Similarly, `Seed` is converted to `Long_Long_Long_Integer`.

When we use the `Image` attribute, the compiler automatically selects a numeric type which has a suitable range for the named number. In the example above, we wouldn't be able to represent the value of `Seed` with **Integer**, so the compiler selected `Long_Long_Long_Integer`. Of course, we could have also specified the type by using explicit *type conversions* (page 319) or a *qualified expressions* (page 338):

Listing 876: show_static_expressions.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Static_Expressions;
4 use Static_Expressions;
5
6 procedure Show_Static_Expressions is
7 begin
8   Put_Line (Long_Long_Float (Pi)'Image);
9   Put_Line (Long_Long_Float' (Pi)'Image);
10 end Show_Static_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↳ Conversion_To_Non_Universal_Types
MD5: 18bcc3bffd51ebc1bc98976ed1597f01

Runtime output

```
3.14159265358979324E+00
3.14159265358979324E+00
```

Now, we're explicitly converting to **Long_Long_Float** in the first call to Put_Line and using a qualified expression in the second call to Put_Line.

A conversion is also performed when we use a named number in an object declaration:

Listing 877: show_static_expressions.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Static_Expressions;
4 use Static_Expressions;
5
6 procedure Show_Static_Expressions is
7   Two_Pi : constant Float := 2.0 * Pi;
8   -- Same as:
9   -- Two_Pi: constant Float :=
10  --      2.0 * Float (Pi);
11
12   Two_Pi_More_Precise :
13     constant Long_Long_Float := 2.0 * Pi;
14   -- Same as:
15   -- Two_Pi_More_Precise :
16   --     constant Long_Long_Float :=
17   --         2.0 * Long_Long_Float (Pi);
18 begin
19   Put_Line (Two_Pi'Image);
20   Put_Line (Two_Pi_More_Precise'Image);
21 end Show_Static_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↳ Conversion_To_Non_Universal_Types
MD5: c918cdcb4e927cfbc1fbe6ffb0277178

Runtime output

```
6.28319E+00
6.28318530717958648E+00
```

In this example, Pi is converted to **Float** in the declaration of `Two_Pi` because we use the **Float** type in its declaration. Likewise, Pi is converted to **Long_Long_Float** in the declaration of `Two_Pi_More_Precise` because we use the **Long_Long_Float** type in its declaration. (Actually, the same conversion is performed for each instance of the real literal `2.0` in this example.)

Note that the range of the type we select might not be suitable for the named number we want to use. For example:

Listing 878: `show_static_expressions.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Static_Expressions;
4 use Static_Expressions;
5
6 procedure Show_Static_Expressions is
7     Initial_Seed : constant
8         Long_Long_Long_Integer :=
9         Super_Seed;
10 begin
11     Put_Line (Initial_Seed'Image);
12 end Show_Static_Expressions;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
 ↪ Conversion_To_Non_Universal_Types
 MD5: 2f8e26fbcd0b5defd94ffef570c0f087

Build output

```
show_static_expressions.adb:9:08: error: value not in range of type "Standard.Long_
↪ Long_Long_Integer"
show_static_expressions.adb:9:08: error: static expression fails Constraint_Check
gprbuild: *** compilation phase failed
```

In this example, we get a compilation error because the range of the `Long_Long_Long_Integer` type isn't enough to store the value of the `Super_Seed`.

For further reading...

To circumvent the compilation error in the code example we've just seen, the best alternative is to use *big numbers* (page 683) — we discuss this topic later on in this chapter:

Listing 879: `show_static_expressions.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use Ada.Numerics.Big_Numbers.Big_Integers;
5
6 with Static_Expressions;
7 use Static_Expressions;
8
9 procedure Show_Static_Expressions is
10     Initial_Seed : constant
11         Big_Integer :=
12         Super_Seed;
```



```

13 begin
14   Put_Line (Initial_Seed'Image);
15 end Show_Static_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Numeric_Types.
↳ Conversion_To_Non_Universal_Types
MD5: bf1511f1b8bf3965baa86b953c56c406

Runtime output

```

↳ 8759293341409421973222546428841660585569956482050794686013387595655014366454466435608186698777

```

By changing the type from `Long_Long_Long_Integer` to `Big_Integer`, we get rid of the compilation error. (The value of `Super_Seed` — stored in `Initial_Seed` — is displayed at runtime.)

Universal Fixed

For fixed-point types, we also have a corresponding universal type. However, in contrast to the universal real and integer types, universal fixed types aren't an abstraction used in static expressions, but rather a concept that permeates actual fixed-point types. In fact, for *fixed-point types* (page 717), some operations are accomplished via universal fixed types — for example, the conversion between fixed-point types and the multiplication and division operations.

Let's start by analyzing how floating-point and integer types associate their operations to the specific type of an object. For example, if we have an object `A` of type `Float` in a multiplication, we cannot just write `A * B` if we want to multiply `A` by an object `B` of another floating-point type — if `B` is of type `Long_Float`, for example, writing `A * B` triggers a compilation error. (Otherwise, which precision should be used for the result?) Therefore, we have to convert one of the objects to have matching types:

Listing 880: `show_float_multiplication_mismatch.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Float_Multiplication_Mismatch is
4   F : Float := 0.25;
5   LF : Long_Float := 0.50;
6 begin
7   F := F * LF;
8   Put_Line ("F = " & F'Image);
9 end Show_Float_Multiplication_Mismatch;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Float_
↳ Multiplication
MD5: 88ce3a0f29e2bd31ddfc491557d7f0e3

Build output

```

show_float_multiplication_mismatch.adb:7:11: error: invalid operand types for
↳ operator "*"
show_float_multiplication_mismatch.adb:7:11: error: left operand has type
↳ "Standard.Float"
show_float_multiplication_mismatch.adb:7:11: error: right operand has type
↳ "Standard.Long_Float"
gprbuild: *** compilation phase failed

```

This code example fails to compile because of the $F * LF$ operation. (We could correct the code by writing $F * \text{Float } (LF)$, for example.)

In contrast, for fixed-point types, we can mix objects of different types in a multiplication or division. (In this case, mixing is allowed for the convenience of the programmer.) For example:

Listing 881: normalized_fixed_point_types.ads

```

1 package Normalized_Fixed_Point_Types is
2
3     type TQ31 is
4         delta 2.0 ** (-31)
5         range -1.0 .. 1.0 - 2.0 ** (-31);
6
7     type TQ15 is
8         delta 2.0 ** (-15)
9         range -1.0 .. 1.0 - 2.0 ** (-15);
10
11 end Normalized_Fixed_Point_Types;
```

Listing 882: show_fixed_multiplication.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Normalized_Fixed_Point_Types;
4 use Normalized_Fixed_Point_Types;
5
6 procedure Show_Fixed_Multiplication is
7     A : TQ15 := 0.25;
8     B : TQ31 := 0.50;
9 begin
10     A := A * B;
11     Put_Line ("A = " & A'Image);
12 end Show_Fixed_Multiplication;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Fixed_Point_Multiplication
MD5: a4cefdc29a562fbec30b6864b6ec2602

Runtime output

A = 0.12500

In this example, the $A * B$ is accepted by the compiler, even though A and B have different types. This is only possible because the multiplication operation of fixed-point types makes use of the universal fixed type. In other words, the multiplication operation in this code example doesn't operate directly on the fixed-point type TQ31. Instead, it converts A and B to the universal fixed type, performs the operation using this type, and converts back to the original type — TQ15 in this case.

In addition to the multiplication operation, other operations such as the conversion between fixed-point types and the division operations make use of universal fixed types:

Listing 883: custom_decimal_types.ads

```

1 package Custom_Decimal_Types is
2
3     type T3_D3 is delta 10.0 ** (-3) digits 3;
4     type T3_D6 is delta 10.0 ** (-3) digits 6;
```

(continues on next page)

(continued from previous page)

```

5  type T6_D6 is delta 10.0 ** (-6) digits 6;
6
7  end Custom_Decimal_Types;

```

Listing 884: show_universal_fixed.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Custom_Decimal_Types;
4  use Custom_Decimal_Types;
5
6  procedure Show_Universal_Fixed is
7      Val_T3_D3 : T3_D3;
8      Val_T3_D6 : T3_D6;
9      Val_T6_D6 : T6_D6;
10 begin
11     Val_T3_D3 := 0.65;
12
13     Val_T3_D6 := T3_D6 (Val_T3_D3);
14     -- ~~~~~
15     --      type conversion using
16     --      universal fixed type
17
18     Val_T6_D6 := T6_D6 (Val_T3_D6);
19     -- ~~~~~
20     --      type conversion using
21     --      universal fixed type
22
23     Put_Line ("Val_T3_D3 = "
24              & Val_T3_D3'Image);
25     Put_Line ("Val_T3_D6 = "
26              & Val_T3_D6'Image);
27     Put_Line ("Val_T6_D6 = "
28              & Val_T3_D6'Image);
29     Put_Line ("-----");
30
31     Val_T3_D6 := Val_T6_D6 * 2.0;
32     -- ~~~~~
33     --      using universal fixed type for
34     --      the multiplication operation
35     Put_Line ("Val_T3_D6 = "
36              & Val_T3_D6'Image);
37
38     Val_T3_D6 := Val_T6_D6 / Val_T3_D3;
39     -- ~~~~~
40     --      different fixed-point types:
41     --      using universal fixed type for
42     --      the division operation
43     Put_Line ("Val_T3_D6 = "
44              & Val_T3_D6'Image);
45
46 end Show_Universal_Fixed;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Universal_Fixed
MD5: 1e253d8a39576f817b2130aa35929d96

Runtime output

```

Val_T3_D3 = 0.650
Val_T3_D6 = 0.650
Val_T6_D6 = 0.650
-----
Val_T3_D6 = 1.300
Val_T3_D6 = 1.000

```

In this example, the conversion from the fixed-point type T3_D3 to the T3_D6 and T6_D6 types is performed via universal fixed types.

Similarly, the multiplication operation `Val_T6_D6 * 2.0` uses universal fixed types. Here, we're actually multiplying a variable of type T6_D6 by two and assigning it to a variable of type Val_T3_D6. Although these variables have different fixed-point types, no explicit conversion (e.g.: `Val_T3_D6 := T3_D6 (Val_T6_D6 * 2.0);`) is required in this case because the result of the operation is of universal fixed type, so that it can be assigned to a variable of any fixed-point type.

Finally, in the `Val_T3_D6 := Val_T6_D6 / Val_T3_D3` statement, we're using three fixed-point types: we're dividing a variable of type T6_D6 by a variable of type T3_D3, and assigning it to a variable of type T3_D6. All these operations are only possible without explicit type conversions because the underlying types for the fixed-point division operation are universal fixed types.

For further reading...

It's possible to implement custom `*` and `/` operators for fixed-point types. However, those operators do **not** override the corresponding operators for universal fixed types. For example:

Listing 885: `normalized_fixed_point_types.ads`

```

1 package Normalized_Fixed_Point_Types is
2
3   type TQ63 is
4     delta 2.0 ** (-63)
5     range -1.0 .. 1.0 - 2.0 ** (-63);
6
7   type TQ31 is
8     delta 2.0 ** (-31)
9     range -1.0 .. 1.0 - 2.0 ** (-31);
10
11  overriding
12  -- ^^^^^^
13  -- "+" operator is overriding!
14  function "+" (L, R : TQ31)
15    return TQ31;
16
17  not overriding
18  -- ^^^^^^^^^
19  -- "*" operator is NOT overriding!
20  function "*" (L, R : TQ31)
21    return TQ31;
22
23  type TQ15 is
24    delta 2.0 ** (-15)
25    range -1.0 .. 1.0 - 2.0 ** (-15);
26
27 end Normalized_Fixed_Point_Types;

```

Listing 886: normalized_fixed_point_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Normalized_Fixed_Point_Types is
4
5     function "+" (L, R : TQ31)
6         return TQ31 is
7     begin
8         Put_Line
9             ("=> Overriding '+'");
10        return TQ31 (TQ63 (L) + TQ63 (R));
11    end "+";
12
13    function "*" (L, R : TQ31)
14        return TQ31 is
15    begin
16        Put_Line
17            ("=> Custom "
18            & "non-overriding '*'");
19        return TQ31 (TQ63 (L) * TQ63 (R));
20    end "*";
21
22 end Normalized_Fixed_Point_Types;

```

Listing 887: show_fixed_multiplication.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Normalized_Fixed_Point_Types;
4 use Normalized_Fixed_Point_Types;
5
6 procedure Show_Fixed_Multiplication is
7     Q31_A : TQ31 := 0.25;
8     Q31_B : TQ31 := 0.50;
9     Q15_A : TQ15 := 0.25;
10    Q15_B : TQ15 := 0.50;
11 begin
12     Q31_A := Q31_A * Q31_B;
13     Put_Line ("Q31_A = " & Q31_A'Image);
14
15     Q15_A := Q15_A * Q15_B;
16     Put_Line ("Q15_A = " & Q15_A'Image);
17
18     Q15_A := TQ15 (Q31_A) * Q15_B;
19     --
20     -- A conversion is required because of
21     -- the multiplication operator of
22     -- TQ15.
23     Put_Line ("Q31_A = " & Q31_A'Image);
24 end Show_Fixed_Multiplication;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Universal_Types.Fixed_Point_
 Custom_Multiplication
 MD5: 954ada297ac676ab1f11447083d87882

Runtime output

```

=> Custom non-overriding '*'
Q31_A = 0.1250000000
Q15_A = 0.1250000000
Q31_A = 0.1250000000

```

In this example, we're declaring a custom multiplication operator for the TQ31 type. As we can see in the declaration, we specify that it's **not overriding** the `*` operator. (Removing the **not** keyword triggers a compilation error.) In contrast, for the `+` operator, we're indeed overriding the default `+` operator of the TQ31 type in the `Normalized_Fixed_Point_Types` because the addition operator is associated with its corresponding fixed-point type, not with the universal fixed type. In the `Q31_A := Q31_A * Q31_B` statement, we see at runtime (through the `"=> Custom non-overriding '*'"` message) that the custom multiplication is being used.

However, because of this custom `*` operator, we cannot mix objects of this type with objects of other fixed-point types in multiplication or division operations. Therefore, for a statement such as `Q15_A := Q31_A * Q15_B`, we have to convert `Q31_A` to the TQ15 type before multiplying it by `Q15_B`.

In the Ada Reference Manual

- 4.5.5 Multiplying Operators¹⁴⁹

25.8.3 Base types

You might remember our discussion on *root types* (page 303) and the corresponding numeric root types.

Ada also has the concept of base types, which *sounds* similar to the concept of the root type. However, the focus of each one is different: while the root type refers to the derivation tree of a type, the base type refers to the constraints of a type.

In fact, the base type denotes the unconstrained underlying hardware representation selected for a given numeric type. For example, if we were making use of a constrained type `T`, the compiler would select a type based on the hardware characteristics that has sufficient precision to represent `T` on the target platform. Of course, that type — the base type — would necessarily be unconstrained.

Let's discuss the **Integer** type as an example. The Ada standard specifies that the minimum range of the **Integer** type is $-2^{15} + 1 \dots 2^{15} - 1$. In modern 64-bit systems — where wider types such as **Long_Integer** are defined — the range is at least $-2^{31} + 1 \dots 2^{31} - 1$. Therefore, we could think of the **Integer** type as having the following declaration:

```
type Integer is
  range -2 ** 31 .. 2 ** 31 - 1;
```

However, even though **Integer** is a predefined Ada type, it's actually a subtype of an anonymous type. That anonymous "type" is the hardware's representation for the numeric type as chosen by the compiler based on the requested range (for the signed integer types) or digits of precision (for floating-point types). In other words, these types are actually subtypes of something that does not have a specific name in Ada, and that is not constrained.

In effect,

```
type Integer is
  range -2 ** 31 .. 2 ** 31 - 1;
```

is really as if we said this:

¹⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-5.html>

```
subtype Integer is
  Some_Hardware_Type_With_Sufficient_Range
  range -2 ** 31 .. 2 ** 31 - 1;
```

Since the `Some_Hardware_Type_With_Sufficient_Range` type is anonymous and we therefore cannot refer to it in the code, we just say that **Integer** is a type rather than a subtype.

Let's focus on signed integers — as the other numerics work the same way. When we declare a signed integer type, we have to specify the required range, statically. If the compiler cannot find a hardware-defined or supported signed integer type with at least the range requested, the compilation is rejected. For example, in current architectures, the code below most likely won't compile:

Listing 888: int_def.ads

```
1 package Int_Def is
2
3     type Too_Big_To_Fail is
4       range -2 ** 255 .. 2 ** 255 - 1;
5
6 end Int_Def;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Base_Type.Very_Big_Range
MD5: 60c2b9219a0cf080997707a99a0b162b

Build output

```
int_def.ads:4:07: error: integer type definition bounds out of range
gprbuild: *** compilation phase failed
```

Otherwise, the compiler maps the named Ada type to the hardware "type", presumably choosing the smallest one that supports the requested range. (That's why the range has to be static in the source code, unlike for explicit subtypes.)

Base

The `Base` attribute gives us the unconstrained underlying hardware representation selected for a given numeric type. As an example, let's say we declared a subtype of the **Integer** type named `One_To_Ten`:

Listing 889: my_integers.ads

```
1 package My_Integers is
2
3     subtype One_To_Ten is Integer
4       range 1 .. 10;
5
6 end My_Integers;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Base_Type.Base_Attr
MD5: e3f8310ed742e61a65728fecb6caa557

If we then use the `Base` attribute — by writing `One_To_Ten'Base` —, we're actually referring to the unconstrained underlying hardware representation selected for `One_To_Ten`. As `One_To_Ten` is a subtype of the **Integer** type, this also means that `One_To_Ten'Base` is equivalent to `Integer'Base`, i.e. they refer to the same base type. (This base type is the

underlying hardware type representing the **Integer** type — but is not the **Integer** type itself.)

The following example shows how the Base attribute affects the bounds of a variable:

Listing 890: show_base.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with My_Integers; use My_Integers;
3
4  procedure Show_Base is
5      C : constant One_To_Ten := One_To_Ten'Last;
6  begin
7      Using_Constrained_Subtype : declare
8          V : One_To_Ten := C;
9      begin
10         Put_Line
11             ("Increasing value for One_To_Ten...");
12
13         V := One_To_Ten'Succ (V);
14     exception
15         when others =>
16             Put_Line ("Exception raised!");
17     end Using_Constrained_Subtype;
18
19     Using_Base : declare
20         V : One_To_Ten'Base := C;
21     begin
22         Put_Line
23             ("Increasing value for One_To_Ten'Base...");
24
25         V := One_To_Ten'Succ (V);
26     exception
27         when others =>
28             Put_Line ("Exception raised!");
29     end Using_Base;
30
31     Put_Line ("One_To_Ten'Last: "
32             & One_To_Ten'Last'Image);
33     Put_Line ("One_To_Ten'Base'Last: "
34             & One_To_Ten'Base'Last'Image);
35 end Show_Base;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Base_Type.Base_Attr
MD5: ce3e9fb3ff1619e835e9108ae0a787e7

Build output

```

show_base.adb:13:22: warning: value not in range of type "One_To_Ten" defined at
↳my_integers.ads:3 [enabled by default]
show_base.adb:13:22: warning: Constraint_Error will be raised at run time [enabled
↳by default]
```

Runtime output

```

Increasing value for One_To_Ten...
Exception raised!
Increasing value for One_To_Ten'Base...
One_To_Ten'Last: 10
One_To_Ten'Base'Last: 2147483647
```

In the first block of the example (Using_Constrained_Subtype), we're asking for the next

value after the last value of a range — in this case, `One_To_Ten'Succ (One_To_Ten'Last)`. As expected, since the last value of the range doesn't have a successor, a constraint exception is raised.

In the `Using_Base` block, we're declaring a variable `V` of `One_To_Ten'Base` subtype. In this case, the next value exists — because the condition `One_To_Ten'Last + 1 <= One_To_Ten'Base'Last` is true —, so we can use the `Succ` attribute without having an exception being raised.

In the following example, we adjust the result of additions and subtractions to avoid constraint errors:

Listing 891: `my_integers.ads`

```
1 package My_Integers is
2
3     subtype One_To_Ten is Integer range 1 .. 10;
4
5     function Sat_Add (V1, V2 : One_To_Ten'Base)
6         return One_To_Ten;
7
8     function Sat_Sub (V1, V2 : One_To_Ten'Base)
9         return One_To_Ten;
10
11 end My_Integers;
```

Listing 892: `my_integers.adb`

```
1 -- with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Integers is
4
5     function Saturate (V : One_To_Ten'Base)
6         return One_To_Ten is
7     begin
8         -- Put_Line ("SATURATE " & V'Image);
9
10        if V < One_To_Ten'First then
11            return One_To_Ten'First;
12        elsif V > One_To_Ten'Last then
13            return One_To_Ten'Last;
14        else
15            return V;
16        end if;
17    end Saturate;
18
19    function Sat_Add (V1, V2 : One_To_Ten'Base)
20        return One_To_Ten is
21    begin
22        return Saturate (V1 + V2);
23    end Sat_Add;
24
25    function Sat_Sub (V1, V2 : One_To_Ten'Base)
26        return One_To_Ten is
27    begin
28        return Saturate (V1 - V2);
29    end Sat_Sub;
30
31 end My_Integers;
```

Listing 893: show_base.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with My_Integers; use My_Integers;
3
4 procedure Show_Base is
5
6     type Display_Saturate_Op is (Add, Sub);
7
8     procedure Display_Saturate
9         (V1, V2 : One_To_Ten;
10          Op     : Display_Saturate_Op)
11     is
12         Res : One_To_Ten;
13     begin
14         case Op is
15             when Add =>
16                 Res := Sat_Add (V1, V2);
17             when Sub =>
18                 Res := Sat_Sub (V1, V2);
19             end case;
20         Put_Line ("SATURATE " & Op'Image
21                  & " (" & V1'Image
22                  & ", " & V2'Image
23                  & ") = " & Res'Image);
24     end Display_Saturate;
25
26 begin
27     Display_Saturate (1, 1, Add);
28     Display_Saturate (10, 8, Add);
29     Display_Saturate (1, 8, Sub);
30 end Show_Base;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Base_Type.Base_Attr_Sat
MD5: e9b31345c2efc056bdb71824072852d0

Runtime output

```

SATURATE ADD ( 1, 1) = 2
SATURATE ADD ( 10, 8) = 10
SATURATE SUB ( 1, 8) = 1

```

In this example, we're using the Base attribute to declare the parameters of the Sat_Add, Sat_Sub and Saturate functions. Note that the parameters of the Display_Saturate procedure are of One_To_Ten type, while the parameters of the Sat_Add, Sat_Sub and Saturate functions are of the (unconstrained) base subtype (One_To_Ten'Base). In those functions, we perform operations using the parameters of unconstrained subtype and adjust the result — in the Saturate function — before returning it as a constrained value of One_To_Ten subtype.

The code in the body of the My_Integers package contains lines that were commented out — to be more precise, a call to Put_Line call in the Saturate function. If you uncomment them, you'll see the value of the input parameter V (of One_To_Ten'Base type) in the runtime output of the program before it's adapted to fit the constraints of the One_To_Ten subtype.

25.8.4 Discrete and Real Numeric Types

Discrete Numeric Types

In the Introduction to Ada course, we've seen that Ada has two kinds of discrete numeric types: *signed integer* (page 47) and *modular* (page 50) types. For example:

Listing 894: num_types.ads

```

1 package Num_Types is
2
3     type Signed_Integer is range 1 .. 1_000_000;
4     type Modular is mod 2**32;
5
6 end Num_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Discrete_And_Real_Numeric_Types.
 ↳ Signed_Modular_Types
 MD5: 2dff9fe22c6bbe52f964befccf68debf

Remember that modular types are similar to *unsigned* integer types in other programming languages.

In this chapter, we review these types and look into a couple of details that haven't been covered yet. We start the discussion with *signed integer types* (page 649), and then move on to *modular types* (page 657).

Real Numeric Types

In the Introduction to Ada course, we talked about *floating-point* (page 52) and *fixed-point* (page 107) types. In Ada, these two categories of numeric types belong to the so-called *real types*. In very simple terms, we could say that real types are the ones whose objects we could assign *real numeric literals* (page 622) to. For example:

Listing 895: show_real_numeric_object.adb

```

1 procedure Show_Real_Numeric_Object is
2     V : Float;
3 begin
4     V := 2.3333333333;
5     -- ~~~~~
6     -- real numeric literal
7 end Show_Real_Numeric_Object;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Discrete_And_Real_Numeric_Types.
 ↳ Universal_And_Real_Numeric_Types
 MD5: 58814135a1c545c10f04c86461573c88

Note that we shouldn't confuse real numeric types with *universal real types* (page 629). Even though we can assign a named number of universal real type to an object of a real type, these terms refer to very distinct concepts. For example:

Listing 896: universal_and_real_numeric_types.ads

```

1 package Universal_And_Real_Numeric_Types is
2
3     Pi : constant := 3.1415926535;
4     -- ~~~~~
```

(continues on next page)

(continued from previous page)

```

5      --          universal real type
6
7      V : Float      := Pi;
8      ^^^^^
9      -- real type
10     -- (floating-point type)
11     --
12
13 end Universal_And_Real_Numeric_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Discrete_And_Real_Numeric_Types.
 ↳ Universal_And_Real_Numeric_Types
 MD5: 16860adf08f24d88b49487445a007adb

In this example, `Pi` is a named number of universal real type, while `V` is an object of real type — and of floating-point type, to be more precise.

Note that both real types and universal real types are implicitly derived from the *root real type* (page 303), which we already discussed in another chapter.

i In the Ada Reference Manual

- 3.5.6 Real Types¹⁵⁰

25.8.5 Integer types

In the *Introduction to Ada* (page 47) course, we mentioned that you can define your own integer types in Ada. In fact, typically you're expected to do so, as Ada only guarantees the existence of a single integer type — and the names of a few optional integer types. Even though a specific compiler might offer multiple predefined integer types, there's no guarantee that it does that. Therefore, you should carefully evaluate the expected range of each integer type in your implementation and specify that information in the corresponding type definition.

i In the Ada Reference Manual

- 3.5.4 Integer Types¹⁵¹

Predefined integer types

Ada only has a single predefined integer type (**Integer**) and two subtypes (**Natural** and **Positive**). Although the actual range of **Integer** depends on the compiler and the target architecture, it must at least support a 16-bit range — we can say that the following specification is the minimum requirement for the **Integer** type:

```

package Standard is
    -- [...]

    type Integer is
        range -2**15 + 1 .. +2**15 - 1;

```

(continues on next page)

¹⁵⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-6.html>

¹⁵¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html>

(continued from previous page)

```

subtype Natural is Integer
  range 0 .. Integer'Last;

subtype Positive is Integer
  range 1 .. Integer'Last;

-- [...]

end Standard;
```

Note that the range of **Integer** doesn't start at -2^{15} , but rather at $-2^{15} + 1$, which might seem a bit unusual. Thus, if your algorithm requires the existence of -2^{15} , you have a good reason to define a custom range instead of relying on the **Integer** type.

As we've just said, the Ada standard only guarantees that **Integer** is at least a 16-bit type, but it doesn't define its actual range for a specific compiler or target architecture. For example, **Integer** could be defined as a 32-bit type:

Listing 897: check_integer_type_range.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Check_Integer_Type_Range is
4 begin
5   Put_Line ("Integer'Size : "
6             & Integer'Size'Image
7             & " bits");
8   Put_Line ("Integer'First : "
9             & Integer'First'Image);
10  Put_Line ("Integer'Last : "
11            & Integer'Last'Image);
12 end Check_Integer_Type_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Check_Integer_Type_Range
 MD5: a883bbe5aca86a924aa1300736232284

Runtime output

```

Integer'Size : 32 bits
Integer'First : -2147483648
Integer'Last  : 2147483647
```

When running the example above on a typical PC, we might indeed confirm that **Integer** is a 32-bit type — ranging from -2147483648 up to 2147483647. Of course, this doesn't go against the Ada standard, as it doesn't specify the maximum range of the **Integer** type, only the minimum range.

The Ada standard also recommends that the **Long_Integer** type should be available if the target architecture supports at least 32-bit operations. However, the standard only guarantees that, if the **Long_Integer** is available, it must support at least a 32-bit range — again, starting at $-2^{31} + 1$ instead of -2^{31} :

```

package Standard is

  -- [...]

  type Long_Integer is
```

(continues on next page)

(continued from previous page)

```

    range -2**31 + 1 .. +2**31 - 1;

    -- [...]

end Standard;
```

Since this is a minimum requirement, it is possible that different types have the same range — e.g. **Integer** and **Long_Integer** could have the same range on a specific target architecture.

In addition, the Ada standard suggests that compilers may offer integer types with names such as **Long_Long_Integer** and **Long_Long_Long_Integer** — or **Short_Integer** and **Short_Short_Integer**. However, all these types are considered non-portable, as there's no requirement concerning their availability or expected range.

In other languages

In C, you have a longer list of standard integer types:

Listing 898: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      printf("signed char:  %zu bytes\n",
6             sizeof(signed char) * 8);
7      printf("short int:    %zu bytes\n",
8             sizeof(short int) * 8);
9      printf("int:          %zu bytes\n",
10             sizeof(int) * 8);
11     printf("long int:     %zu bytes\n",
12            sizeof(long int) * 8);
13     printf("long long int: %zu bytes\n",
14            sizeof(long long int) * 8);
15     return 0;
16 }
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Integer_Types_C
MD5: 3dcf1d4b70ad7573e5e271b3651200e9

Runtime output

```

signed char:  8 bytes
short int:    16 bytes
int:          32 bytes
long int:     64 bytes
long long int: 64 bytes
```

(Note that some of the types above aren't available in all versions of the C standard.)

For the types above, there are no equivalent types in the Ada standard. (However, a compiler may implement this equivalence for practical reasons.) Therefore, if you're porting code from C to Ada, for example, you should check the expected range of your algorithm and specify the corresponding types in the Ada implementation.

In the GNAT toolchain

The GNAT compiler provides a couple of integer types in addition to the standard **Integer** type:

Listing 899: show_gnat_integer_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_GNAT_Integer_Types is
4 begin
5
6     Put_Line ("Short_Short_Integer'Size:      "
7              & Short_Short_Integer'Size'Image
8              & " bits");
9
10    Put_Line ("Short_Integer'Size:              "
11             & Short_Integer'Size'Image
12             & " bits");
13
14    Put_Line ("Integer'Size:                    "
15             & Integer'Size'Image
16             & " bits");
17
18    Put_Line ("Long_Integer'Size:               "
19             & Long_Integer'Size'Image
20             & " bits");
21
22    Put_Line ("Long_Long_Integer'Size:          "
23             & Long_Long_Integer'Size'Image
24             & " bits");
25
26    Put_Line ("Long_Long_Long_Integer'Size:     "
27             & Long_Long_Long_Integer'Size'Image
28             & " bits");
29
30 end Show_GNAT_Integer_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.GNAT_Integer_Types

MD5: 86c1e4fa1c079dadae1817bfdb5659b1

Runtime output

```

Short_Short_Integer'Size:      8 bits
Short_Integer'Size:           16 bits
Integer'Size:                 32 bits
Long_Integer'Size:            64 bits
Long_Long_Integer'Size:       64 bits
Long_Long_Long_Integer'Size:  128 bits
```

The actual range of each of these integer types depends on the target architecture. (Note that you may have different types with the same range.)

Also, when interfacing with C code, GNAT guarantees the following type equivalence:

C type	Ada type
signed char	Short_Short_Integer
short int	Short_Integer
int	Integer
long	Long_Integer
long long	Long_Long_Integer

Custom integer types

As we've mentioned before, for the language-defined numeric data types such as **Integer** or **Long_Integer**, the range selected by the compiler may not correspond to the required range of the numeric algorithm we're implementing. Therefore, it is best to simply declare custom types with the necessary ranges specified. To do that, you should evaluate the algorithm and reach a clear understanding about the adequate range of each integer type — this should be based on the requirements of the algorithm.

For example, if some coefficients in your algorithm expected at least 32-bit precision, you may consider defining this type:

Listing 900: custom_integer_types.ads

```

1 package Custom_Integer_Types is
2
3     type Coefficient is
4         range -2**31 .. +2**31 - 1;
5
6     -- [...]
7
8 end Custom_Integer_Types;
```

Listing 901: show_custom_integer_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Integer_Types;
4 use Custom_Integer_Types;
5
6 procedure Show_Custom_Integer_Types is
7 begin
8     Put_Line ("Coefficient'Size  :\"
9              & Coefficient'Size'Image
10             & " bits");
11     Put_Line ("Coefficient'First  :\"
12              & Coefficient'First'Image);
13     Put_Line ("Coefficient'Last   :\"
14              & Coefficient'Last'Image);
15 end Show_Custom_Integer_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Custom_Integer_Type
MD5: 44284e0eb7c1e37fbc2b4a969fd0928f

Runtime output

```

Coefficient'Size  : 32 bits
Coefficient'First  :-2147483648
Coefficient'Last   : 2147483647
```

In this example, we declare the 32-bit Coefficient type. We ensure that it's a 32-bit type by explicitly writing **range -2**31 .. +2**31 - 1**.

Note that a custom type definition is always derived from the *root integer type* (page 303), which we discussed in another chapter.

Illegal integer definitions

If the specified range cannot be supported by the target machine, the Ada compiler will reject the source code containing the type declaration (and all clients of that code). For example:

Listing 902: custom_integer_types.ads

```
1 package Custom_Integer_Types is
2
3     type Int_1024_Bits is
4         range -2**1023 .. +2**1023 - 1;
5
6     -- [...]
7
8 end Custom_Integer_Types;
```

Listing 903: show_custom_integer_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Integer_Types;
4 use Custom_Integer_Types;
5
6 procedure Show_Custom_Integer_Types is
7 begin
8     Put_Line ("Int_1024_Bits'Size : "
9             & Int_1024_Bits'Size'Image
10            & " bits");
11     Put_Line ("Int_1024_Bits'First : "
12             & Int_1024_Bits'First'Image);
13     Put_Line ("Int_1024_Bits'Last : "
14             & Int_1024_Bits'Last'Image);
15 end Show_Custom_Integer_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Illegal_Custom_Integer_Type
↪ Integer_Type
MD5: 8b17662d3ceb8bc91fdb95006170dd97

Build output

```
custom_integer_types.ads:4:06: error: integer type definition bounds out of range
gprbuild: *** compilation phase failed
```

In this example, we're trying to define a 1024-bit integer type. Unless you're compiling this code example many decades in the future, the compiler will (most likely) reject this definition because current hardware architectures don't support this range in any way. In order to handle integer values in such ranges, you might consider using *big numbers* (page 683).

You can query the maximum supported range by using the `System.Min_Int` and `System.Max_Int` values. We discuss this topic *next* (page 655).

In the GNAT toolchain

As of 2025, GNAT supports 128-bit integers:

Listing 904: custom_integer_types.ads

```

1 package Custom_Integer_Types is
2
3     type Int_128_Bits is
4         range -2**127 .. +2**127 - 1;
5
6     -- [...]
7
8 end Custom_Integer_Types;
```

Listing 905: show_custom_integer_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Integer_Types;
4 use Custom_Integer_Types;
5
6 procedure Show_Custom_Integer_Types is
7 begin
8     Put_Line ("Int_128_Bits'Size : "
9               & Int_128_Bits'Size'Image
10              & " bits");
11     Put_Line ("Int_128_Bits'First : "
12               & Int_128_Bits'First'Image);
13     Put_Line ("Int_128_Bits'Last : "
14               & Int_128_Bits'Last'Image);
15 end Show_Custom_Integer_Types;
```

For further reading...

This is a different approach to portability than that of, say, C, where for example type `int` is always defined and hence the client code always compiles, but won't necessarily work at run-time. In that case, at best you find the problem during testing, which is comparatively expensive. Worse, if you don't find out until after deployment, the cost to fix it is much, much higher.

In contrast, with a user-specified integer type, if the specified range cannot be supported by the (perhaps new) target machine, you find out at compile-time, which is far less expensive and more robust too.

System max. and min. values

As we've just mentioned, a custom type definition is derived from the *root integer type* (page 303). The base range of the root integer type is `System.Min_Int .. System.Max_Int`.

The value of `System.Min_Int` and `System.Max_Int` depends on the target system. For example:

Listing 906: show_system_int_range.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 procedure Show_System_Int_Range is
5 begin
6     Put_Line ("System.Min_Int : "
7               & System.Min_Int'Image);
8     Put_Line ("System.Max_Int : "
```

(continues on next page)

(continued from previous page)

```
9         & System.Max_Int'Image);  
10 end Show_System_Int_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.System_Int_Range
MD5: 7919df1c3496592666386d6272ff600d

Runtime output

```
System.Min_Int : -170141183460469231731687303715884105728  
System.Max_Int : 170141183460469231731687303715884105727
```

On a typical desktop PC, you might get the following values:

- System.Min_Int: -170141183460469231731687303715884105728
- System.Max_Int: 170141183460469231731687303715884105727

Because *custom integer types* (page 653) are implicitly derived from the root integer type, we cannot declare a custom integer type outside of the System.Min_Int .. System.Max_Int range:

Listing 907: custom_int_out_of_range.ads

```
1 with System;  
2  
3 package Custom_Int_Out_Of_Range is  
4  
5     type Custom_Int is  
6         range System.Min_Int - 1 ..  
7             System.Max_Int + 1;  
8  
9 end Custom_Int_Out_Of_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Custom_Int_Out_Of_Range
MD5: 8f7a58a8097f4caf521338c6a3f90dd0

Build output

```
custom_int_out_of_range.ads:6:06: error: integer type definition bounds out of range  
gprbuild: *** compilation phase failed
```

The compilation of this package fails because the Custom_Int'First is below System.Min_Int and Custom_Int'Last is above System.Max_Int.

Range of base type

As we've said before, a custom type definition is derived from the root integer type. The range of its *base type* (page 643), however, is *not* derived from the root integer type, but rather determined by the range of the type specification. For example:

Listing 908: custom_integer_types.ads

```
1 with System;  
2  
3 package Custom_Integer_Types is
```

(continues on next page)

(continued from previous page)

```

4  type Custom_Int is
5      range 1 .. 10;
6
7
8  end Custom_Integer_Types;

```

Listing 909: show_custom_integer_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Custom_Integer_Types;
4  use Custom_Integer_Types;
5
6  procedure Show_Custom_Integer_Types is
7  begin
8      Put_Line ("Custom_Int'Size : "
9                & Custom_Int'Size'Image
10               & " bits");
11      Put_Line ("Custom_Int'First : "
12                & Custom_Int'First'Image);
13      Put_Line ("Custom_Int'Last : "
14                & Custom_Int'Last'Image);
15
16      Put_Line ("Custom_Int'Base'Size : "
17                & Custom_Int'Base'Size'Image);
18      Put_Line ("Custom_Int'Base'First : "
19                & Custom_Int'Base'First'Image);
20      Put_Line ("Custom_Int'Base'Last : "
21                & Custom_Int'Base'Last'Image);
22  end Show_Custom_Integer_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Integer_Types.Custom_Int_Base_
 ↳ Range
 MD5: 60a43ee215e3ed3267ed9250d143d93f

Runtime output

```

Custom_Int'Size : 4 bits
Custom_Int'First : 1
Custom_Int'Last : 10
Custom_Int'Base'Size : 8
Custom_Int'Base'First : -128
Custom_Int'Base'Last : 127

```

On a typical desktop PC, you might see that the range of Custom_Int'Base is -128 .. 127, while the *system max. and min. values* (page 655) we've seen before had a much wider range.

As a reminder, the range of the base type might be wider than the range of the custom integer type we're defining. (We mentioned this earlier on when discussing *base types* (page 643).)

25.8.6 Modular Types

As we've mentioned in the *Introduction to Ada* (page 50) course, modular types are the Ada version of *unsigned* integer types. We declare a modular type by specifying its modulo — by using the **mod** keyword:

Listing 910: modular_types.ads

```
1 package Modular_Types is
2
3     type Modular is mod 2**32;
4
5 end Modular_Types;
```

Listing 911: show_modular_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Modular_Types;
4 use Modular_Types;
5
6 procedure Show_Modular_Types is
7 begin
8     Put_Line ("Modular'Size  : "
9              & Modular'Size'Image
10             & " bits");
11     Put_Line ("Modular'First : "
12             & Modular'First'Image);
13     Put_Line ("Modular'Last  : "
14             & Modular'Last'Image);
15 end Show_Modular_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_Types
MD5: 535263d0b8646d9d2c312fd2880baa01

Runtime output

```
Modular'Size  : 32 bits
Modular'First : 0
Modular'Last  : 4294967295
```

This example declares the 32-bit modular type Modular.

Note that, different from other languages such as C, the modulus need not be a power of two. For example:

Listing 912: modular_types.ads

```
1 package Modular_Types is
2
3     type Modular_10 is mod 10;
4
5 end Modular_Types;
```

Listing 913: show_not_power_of_two_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Modular_Types;
4 use Modular_Types;
5
6 procedure Show_Not_Power_Of_Two_Modular is
7 begin
8     Put_Line ("Modular_10'Size  : "
9             & Modular_10'Size'Image
```

(continues on next page)

(continued from previous page)

```

10         & " bits");
11     Put_Line ("Modular_10'First : "
12             & Modular_10'First'Image);
13     Put_Line ("Modular_10'Last  : "
14             & Modular_10'Last'Image);
15 end Show_Not_Power_Of_Two_Modular;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Modular_Types_Not_Power_Of_Two
 MD5: 624e106f2dfd469d90b940185d8c03e8

Runtime output

```

Modular_10'Size  : 4 bits
Modular_10'First : 0
Modular_10'Last  : 9

```

In this example, the modulus of type `Modular_10` is 10 (which obviously is not a power-of-two number).

There are many attributes on modular types. We talk about them *in another chapter* (page 698).

i In the Ada Reference Manual

- 3.5.4 Integer Types¹⁵²

System max. values for modulus

When we use a power-of-two number as the modulus, the maximum value that we could use in the type declaration is indicated by the `System.Max_Binary_Modulus` constant. In contrast, for non-power-of-two numbers, the maximum value for the modulus is indicated by the `System.Max_Nonbinary_Modulus` constant:

Listing 914: `show_max_binary_nonbinary_modulus.adb`

```

1  with System;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Show_Max_Binary_Nonbinary_Modulus is
5      type Modular_Max is
6          mod System.Max_Binary_Modulus;
7  begin
8      Put_Line
9          ("System.Max_Binary_Modulus - 1 : "
10         & Modular_Max'Last'Image);
11      Put_Line
12          ("System.Max_Nonbinary_Modulus : "
13         & System.Max_Nonbinary_Modulus'Image);
14  end Show_Max_Binary_Nonbinary_Modulus;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Max_Binary_Nonbinary_Modulus_Values
 MD5: f02e2531fafbc8c482e7a7a9bd54df2c

¹⁵² <http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html>

Runtime output

```
System.Max_Binary_Modulus - 1 : 340282366920938463463374607431768211455
System.Max_Nonbinary_Modulus : 4294967295
```

On a typical desktop PC, you might get the following values:

- System.Max_Binary_Modulus: $2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456$
- System.Max_Nonbinary_Modulus: $2^{32} - 1 = 4,294,967,295$

As expected, we can simply use these constants in modular type declarations:

Listing 915: show_max_binary_nonbinary_modulus.ads

```
1 with System;
2
3 package Show_Max_Binary_Nonbinary_Modulus is
4
5     type Modular_Max is
6         mod System.Max_Binary_Modulus;
7
8     type Modular_Max_Non_Power_Two is
9         mod System.Max_Nonbinary_Modulus;
10
11 end Show_Max_Binary_Nonbinary_Modulus;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Modular_Types.Max_Binary_
Nonbinary_Modulus_Types
MD5: 11e96e9def8b005d40e256ebad4be172
```

In this example, we use Max_Binary_Modulus as the modulus of the Modular_Max type, and Max_Nonbinary_Modulus as the modulus of the Modular_Max_Non_Power_Two type.

25.8.7 Floating-point types

In the *Introduction to Ada* (page 52) course, we already covered a couple of details about floating-point types. In this section, we will revise and expand on those topics.

In the Ada Reference Manual

- 3.5.7 Floating Point Types¹⁵³

Decimal precision

The main defining characteristic of a floating-point types is its decimal precision — and not its range, as for integer types. (You may, however, define *ranges for floating-point types* (page 675), as we'll discuss later on.) This means in simple terms that, when the value of a floating-point object of type T is represented as a string, its accuracy is guaranteed for the number of significant decimal digits defined for type T.

For example, consider a number such as 0.123456123, which has 9 significant digits. If we want to store this number in an object with a decimal precision of 6 digits, the number will be *simplified* (actually, truncated) to 0.123456 — which has 6 significant digits:

```
0.123456123    9 significant digits
0.123456       6 significant digits
```

¹⁵³ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-7.html>

Let's see a code example:

Listing 916: show_decimal_digits.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Decimal_Digits is
4
5      type Float_6_Digits is
6          digits 6;
7
8      type Float_9_Digits is
9          digits 9;
10
11     F6 : Float_6_Digits;
12 begin
13     F6 := 0.123456123;
14
15     Put_Line ("F6 = "
16              & F6'Image);
17
18     Put_Line ("F6 = "
19              & Float_9_Digits (F6)'Image
20              & " (9 digits)");
21
22     Put_Line ("Float_6_Digits'Size : "
23              & Float_6_Digits'Size'Image
24              & " bits");
25 end Show_Decimal_Digits;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↳ Point_Decimal_Precision
 MD5: b64d7cce6581a25dd60b932df0816cc9

Runtime output

```

F6 = 1.23456E-01
F6 = 1.23456120E-01 (9 digits)
Float_6_Digits'Size : 32 bits
```

In this example, we define the custom floating-point type `Float_6_Digits`, which has a decimal precision of 6 digits. This ensures that, if we assign a number such as `0.123456123` to a variable `F` of this type, the 6 first significant digits of this number (123456) will be correctly represented. Because these are the only number of digits that the language guarantees, no further digits are used when converting the number to a string — therefore, we see `F6 = 1.23456E+00` in the user message.

However, the digits that we specify in the decimal precision of the type definition are the required *minimum* number of significant decimal digits. This means that the compiler is allowed to make use of a higher precision when storing floating-point values into registers and memory. In fact, the compiler might select a data type that allows for a much higher precision than the one that would be theoretically needed for the decimal precision we requested.

In the code snippet above, we use the `Float_9_Digits (F6)` conversion to display the value stored in `F6` with a decimal precision of 9 digits — the requested precision for the `Float_9_Digits` type. When we display this converted value, we might see (at least, on a desktop PC) that the actual value stored in `F6` isn't 1.23456, but rather a value closer to the one we used in the `F6 := 0.123456123` assignment. This indicates that the underlying hardware precision for the `Float_6_Digits` type is higher than the 6 decimal digits we requested.

Predefined floating-point types

As we know, Ada offers the predefined floating-point type **Float**. If the compiler supports floating-point types with 6 or more digits of decimal precision, then the decimal precision of **Float** must be at least 6 digits:

```
package Standard is
    -- [...]

    type Float is digits 6;

    -- [...]

end Standard;
```

The Ada standard also recommends that, if the **Long_Float** type is made available, its decimal precision must be at least 11 digits:

```
package Standard is
    -- [...]

    type Long_Float is digits 11;

    -- [...]

end Standard;
```

In addition, similar to integer types, the Ada standard suggests that compilers may offer floating-point types with names such as **Long_Long_Float** — or **Short_Float** and **Short_Short_Float**. However, all these types are considered non-portable, as there's no requirement concerning their availability or expected decimal precision.

In other languages

In C, we have a longer list of standard floating-point types:

Listing 917: main.c

```
1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      printf("float:      %zu bytes\n",
6             sizeof(float) * 8);
7      printf("double:     %zu bytes\n",
8             sizeof(double) * 8);
9      printf("long double: %zu bytes\n",
10            sizeof(long double) * 8);
11     return 0;
12 }
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_Point_Types_C

MD5: d284cac03eeae0221fb03cba084852f

Runtime output

```
float:      32 bytes
double:     64 bytes
long double: 128 bytes
```

(Note that some of the types above aren't available in all versions of the C standard.)

For the types above, there are no equivalent types in the Ada standard. (However, a compiler may implement this equivalence for practical reasons.) Therefore, if you're porting code from C to Ada, for example, you should rather check the expected range of your algorithm and specify custom floating-point types in the Ada implementation.

In the GNAT toolchain

The GNAT compiler provides a couple of floating-point types in addition to the standard **Float** type:

Listing 918: show_gnat_float_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_GNAT_Float_Types is
4 begin
5     Put_Line ("Short_Float'Size:          "
6             & Short_Float'Size'Image
7             & " bits");
8
9     Put_Line ("Float'Size:                "
10            & Float'Size'Image
11            & " bits");
12
13     Put_Line ("Long_Float'Size:           "
14            & Long_Float'Size'Image
15            & " bits");
16
17     Put_Line ("Long_Long_Float'Size:      "
18            & Long_Long_Float'Size'Image
19            & " bits");
20
21 end Show_GNAT_Float_Types;
22
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.GNAT_
 Floating_Point_Types
 MD5: dc1c95357172b7795d14b5eb32787ce7

Runtime output

```

Short_Float'Size:      32 bits
Float'Size:           32 bits
Long_Float'Size:      64 bits
Long_Long_Float'Size: 128 bits
```

The actual precision of each of these floating-point types depends on the target architecture. (Note that you may have different types with the same precision.)

Also, when interfacing with C code, GNAT guarantees the following type equivalence:

C type	Ada type
float	Float
double	Long_Float
long double	Long_Long_Float

Custom floating-point types

Similarly to what we discussed for custom integer types, language-defined numeric data types such as **Float** or **Long_Float** may not be sufficient for the requirements of the numeric algorithm we're implementing. So again, it's best to simply declare custom types with sufficient precision. For that, we have to evaluate the algorithm and assess the minimum required precision of each floating-point type — this should be based on the requirements of the algorithm.

For example, if some coefficients from your algorithm expect a decimal precision of at least 12 digits, you may consider defining this type:

Listing 919: custom_floating_point_types.ads

```

1 package Custom_Floating_Point_Types is
2
3     type Coefficient is
4         digits 12;
5
6     -- [...]
7
8 end Custom_Floating_Point_Types;
```

Listing 920: show_custom_floating_point_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Floating_Point_Types;
4 use Custom_Floating_Point_Types;
5
6 procedure Show_Custom_Floating_Point_Types is
7 begin
8     Put_Line ("Coefficient'Digits : "
9             & Coefficient'Digits'Image
10            & " digits");
11     Put_Line ("Coefficient'Size   : "
12             & Coefficient'Size'Image
13            & " bits");
14 end Show_Custom_Floating_Point_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Custom_Floating_Point_Types
MD5: 18ee8dcc851e982cd8f90dd32fa3a413

Runtime output

```

Coefficient'Digits : 12 digits
Coefficient'Size   : 64 bits
```

In this example, we declare the `Coefficient` type with a decimal precision of at least 12 digits. We ensure that this precision is maintained for the type by explicitly writing **digits 12**. (Here, we're using the *Digits attribute* (page 706), which we discuss in another chapter.)

Note that a custom type definition is always derived from the *root real type* (page 303), which we discussed in another chapter.

Derived floating-point types and subtypes

In this section, we have a brief discussion about types derived from floating-point types, as well as subtypes of floating-point types.

Derived floating-point types

As expected, we can derive from any floating-point type. For example:

Listing 921: custom_floating_point_types.ads

```

1 package Custom_Floating_Point_Types is
2
3     type Coefficient is
4       digits 12;
5
6     type Filter_Coefficient is new
7       Coefficient;
8
9 end Custom_Floating_Point_Types;
```

Listing 922: show_derived_floating_point_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Floating_Point_Types;
4 use Custom_Floating_Point_Types;
5
6 procedure Show_Derived_Floating_Point_Types is
7   C : Coefficient;
8   FC : Filter_Coefficient;
9 begin
10  C := 0.532344123;
11  Put_Line ("C = "
12           & C'Image);
13
14  FC := Filter_Coefficient (C);
15  Put_Line ("FC = "
16           & FC'Image);
17 end Show_Derived_Floating_Point_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Derived_Floating_Point_Types
 MD5: 2e777658fa8a7473f4c94f5cc4c4d928

Runtime output

```

C = 5.32344123000E-01
FC = 5.32344123000E-01
```

In this example, we derive the `Filter_Coefficient` type from the `Coefficient` type.

For further reading...

We can also constrain the decimal precision of the derived type. However, this feature is considered obsolescent, so it should be avoided. (Note that this applies to subtypes as well.) For example:

Listing 923: custom_floating_point_types.ads

```

1 package Custom_Floating_Point_Types is
2
3     type Coefficient is
4         digits 12;
5
6     type Filter_Coefficient is new
7         Coefficient
8         digits 6;
9
10 end Custom_Floating_Point_Types;
```

Listing 924: show_derived_floating_point_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Floating_Point_Types;
4 use Custom_Floating_Point_Types;
5
6 procedure Show_Derived_Floating_Point_Types is
7     C : Coefficient;
8     FC : Filter_Coefficient;
9 begin
10    C := 0.532344123;
11    Put_Line ("C = "
12              & C'Image);
13
14    FC := Filter_Coefficient (C);
15    Put_Line ("FC = "
16              & FC'Image);
17 end Show_Derived_Floating_Point_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Derived_Floating_Point_Types
 MD5: 214f4e1097d288186d1bdda1e617ab0d

Runtime output

```
C = 5.32344123000E-01
FC = 5.32344E-01
```

In this example, we derive the `Filter_Coefficient` type from the `Coefficient` type and decrease the decimal precision from 12 to 6 digits.

** In the Ada Reference Manual**

- J.3 Floating Point Types¹⁵⁴

Floating-point subtypes

We can also declare subtypes of floating-point types. For example:

Listing 925: custom_floating_point_types.ads

```

1 package Custom_Floating_Point_Types is
2
3     type Coefficient is
```

(continues on next page)

¹⁵⁴ <http://www.ada-auth.org/standards/22rm/html/RM-J-3.html>

(continued from previous page)

```

4     digits 12;
5
6     subtype Filter_Coefficient is
7         Coefficient;
8
9 end Custom_Floating_Point_Types;

```

Listing 926: show_floating_point_subtypes.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Floating_Point_Types;
4 use Custom_Floating_Point_Types;
5
6 procedure Show_Floating_Point_Subtypes is
7     C : Coefficient;
8     FC : Filter_Coefficient;
9 begin
10    C := 0.532344123;
11    Put_Line ("C = "
12              & C'Image);
13
14    FC := C;
15    Put_Line ("FC = "
16              & FC'Image);
17 end Show_Floating_Point_Subtypes;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪Point_Subtypes
 MD5: 4ef29bc26511f975f9aeeb033054a045

Runtime output

```

C = 5.32344123000E-01
FC = 5.32344123000E-01

```

In this example, we declare `Filter_Coefficient` as a subtype of the `Coefficient` type.

Decimal precision of base type

We discussed *base types* (page 643) earlier on. For floating-point types, the decimal precision of the base type of a T type might be higher than the decimal precision we've specified for type T. For example:

Listing 927: show_base_type_precision.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Base_Type_Precision is
4
5     type Float_3_Digits is
6         digits 3;
7
8 begin
9     Put_Line
10        ("Float_3_Digits'Digits      : "
11          & Float_3_Digits'Digits'Image
12          & " digits");

```

(continues on next page)

(continued from previous page)

```

13
14   Put_Line
15     ("Float_3_Digits'Base'Digits  : "
16      & Float_3_Digits'Base'Digits'Image
17      & " digits");
18 end Show_Base_Type_Precision;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪Point_Base_Type_Precision
 MD5: d19483b76713cec3bc4f89db2b24b6d4

Runtime output

```

Float_3_Digits'Digits      : 3 digits
Float_3_Digits'Base'Digits : 6 digits
```

On a typical desktop PC, you may see that the base type of `Float_3_Digits` has 6 digits, while the `Float_3_Digits` type itself has only 3 digits — as requested in its type declaration.

Size of floating-point types

Notice that the size of the `Float_6_Digits` type from the *first code example* (page 661) was 32 bits. Reducing the number of digits might not have a direct impact on the type's size. In fact, on a typical desktop PC, if we reduce the decimal precision of a type to, say, 3 or 2 digits, the compiler will most probably still select a 32-bit floating-point type for the target platform. For example:

Listing 928: custom_floating_point_types.ads

```

1 package Custom_Floating_Point_Types is
2
3   type Float_1_Digits is
4     digits 1;
5   type Float_2_Digits is
6     digits 2;
7   type Float_3_Digits is
8     digits 3;
9   type Float_4_Digits is
10    digits 4;
11  type Float_5_Digits is
12    digits 5;
13  type Float_6_Digits is
14    digits 6;
15  type Float_7_Digits is
16    digits 7;
17  type Float_8_Digits is
18    digits 8;
19  type Float_9_Digits is
20    digits 9;
21  type Float_10_Digits is
22    digits 10;
23  type Float_11_Digits is
24    digits 11;
25  type Float_12_Digits is
26    digits 12;
27  type Float_13_Digits is
28    digits 13;
29  type Float_14_Digits is
```

(continues on next page)

(continued from previous page)

```

30     digits 14;
31     type Float_15_Digits is
32         digits 15;
33     type Float_16_Digits is
34         digits 16;
35     type Float_17_Digits is
36         digits 17;
37     type Float_18_Digits is
38         digits 18;
39
40 end Custom_Floating_Point_Types;

```

Listing 929: show_decimal_digits.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Custom_Floating_Point_Types;
4  use Custom_Floating_Point_Types;
5
6  procedure Show_Decimal_Digits is
7  begin
8      Put_Line ("Float_1_Digits'Size  :"
9              & Float_1_Digits'Size'Image
10             & " bits");
11      Put_Line ("Float_2_Digits'Size  :"
12              & Float_2_Digits'Size'Image
13             & " bits");
14      Put_Line ("Float_3_Digits'Size  :"
15              & Float_3_Digits'Size'Image
16             & " bits");
17      Put_Line ("Float_4_Digits'Size  :"
18              & Float_4_Digits'Size'Image
19             & " bits");
20      Put_Line ("Float_5_Digits'Size  :"
21              & Float_5_Digits'Size'Image
22             & " bits");
23      Put_Line ("Float_6_Digits'Size  :"
24              & Float_6_Digits'Size'Image
25             & " bits");
26      Put_Line ("Float_7_Digits'Size  :"
27              & Float_7_Digits'Size'Image
28             & " bits");
29      Put_Line ("Float_8_Digits'Size  :"
30              & Float_8_Digits'Size'Image
31             & " bits");
32      Put_Line ("Float_9_Digits'Size  :"
33              & Float_9_Digits'Size'Image
34             & " bits");
35      Put_Line ("Float_10_Digits'Size  :"
36              & Float_10_Digits'Size'Image
37             & " bits");
38      Put_Line ("Float_11_Digits'Size  :"
39              & Float_11_Digits'Size'Image
40             & " bits");
41      Put_Line ("Float_12_Digits'Size  :"
42              & Float_12_Digits'Size'Image
43             & " bits");
44      Put_Line ("Float_13_Digits'Size  :"
45              & Float_13_Digits'Size'Image
46             & " bits");
47      Put_Line ("Float_14_Digits'Size  :");

```

(continues on next page)

(continued from previous page)

```

48         & Float_14_Digits'Size'Image
49         & " bits");
50     Put_Line ("Float_15_Digits'Size : "
51             & Float_15_Digits'Size'Image
52             & " bits");
53     Put_Line ("Float_16_Digits'Size : "
54             & Float_16_Digits'Size'Image
55             & " bits");
56     Put_Line ("Float_17_Digits'Size : "
57             & Float_17_Digits'Size'Image
58             & " bits");
59     Put_Line ("Float_18_Digits'Size : "
60             & Float_18_Digits'Size'Image
61             & " bits");
62 end Show_Decimal_Digits;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪ Point_Decimal_Precision
 MD5: 06ccb5c6a584c6e35328738fd1d693ad

Runtime output

```

Float_1_Digits'Size   : 32 bits
Float_2_Digits'Size   : 32 bits
Float_3_Digits'Size   : 32 bits
Float_4_Digits'Size   : 32 bits
Float_5_Digits'Size   : 32 bits
Float_6_Digits'Size   : 32 bits
Float_7_Digits'Size   : 64 bits
Float_8_Digits'Size   : 64 bits
Float_9_Digits'Size   : 64 bits
Float_10_Digits'Size  : 64 bits
Float_11_Digits'Size  : 64 bits
Float_12_Digits'Size  : 64 bits
Float_13_Digits'Size  : 64 bits
Float_14_Digits'Size  : 64 bits
Float_15_Digits'Size  : 64 bits
Float_16_Digits'Size  : 128 bits
Float_17_Digits'Size  : 128 bits
Float_18_Digits'Size  : 128 bits

```

On a typical desktop PC, we may see the following results:

Min. digits	Max. digits	Size (bits)
1	6	32
7	15	64
16	18	128

Ada doesn't actually give us any guarantees about specific sizes of floating-point data types on the target hardware. However, as you might recall from an earlier chapter, we can request specific sizes for custom types. We discuss this topic next.

Note that, for the example above, the size of the type is equal to the size of its base type, i.e. `Float_1_Digits'Size = Float_1_Digits'Base'Size`, `Float_2_Digits'Size = Float_2_Digits'Base'Size`, and so on.

Custom size of floating-point types

As discussed earlier on, the Ada standard requires that the precision defined after the **digits** keyword of a type is maintained for all objects of that floating-point type. It doesn't require, however, that custom floating-point types — or even predefined floating-point types — have a certain size. Therefore, if we really have to use a specific size for a floating-point data type, we can add the *Size aspect* (page 354) to the type declaration. For example:

Listing 930: show_decimal_digits.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_Digits is
4
5     type Float_6_Digits is
6         digits 6
7         with Size => 128;
8
9 begin
10     Put_Line ("Float_6_Digits'Size  : "
11              & Float_6_Digits'Size'Image
12              & " bits");
13 end Show_Decimal_Digits;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↳ Point_Decimal_Precision
 MD5: 3cfa3768ee1a83904ace653122684c2c

Runtime output

```
Float_6_Digits'Size  : 128 bits
```

In this example, we specify that `Float_6_Digits` requires a size of 128 bits to be represented — instead of the 32 bits that we would typically see on a desktop PC. (Also, remember that this code example won't compile if your target architecture doesn't support 128-bit floating-point data types.)

Range of custom floating-point types and subtypes

In addition to specifying the decimal precision of a floating-point type, we can also specify its range:

Listing 931: show_range_definition.ads

```

1 package Show_Range_Definition is
2
3     type Float_6_Digits_Normalized is
4         digits 6
5         range -1.0 .. 1.0;
6
7 end Show_Range_Definition;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↳ Point_Range_Def
 MD5: 7633f7222b379f9a507736104cb1e883

You probably recall that, for integer types, we were able to declare a type by specifying its range. For floating-point types, however, we cannot specify the floating-point range without

a decimal precision, as the compiler wouldn't be able to infer the intended precision based on the range alone:

Listing 932: show_range_definition.ads

```
1 package Show_Range_Definition is
2
3     type Float_Normalized is
4         range -1.0 .. 1.0;
5         -- ERROR: 'digits' specification
6         --         is missing!
7
8 end Show_Range_Definition;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
↳ Point_Range_Def
MD5: f39353d1bee9793fe973c16aa52bd8d1

Build output

```
show_range_definition.ads:4:12: error: expected an integer type
show_range_definition.ads:4:12: error: found type universal real
show_range_definition.ads:4:20: error: expected an integer type
show_range_definition.ads:4:20: error: found type universal real
gprbuild: *** compilation phase failed
```

Compilation of this code example fails because the decimal precision was not specified. Assigning to objects of different floating-point types works as expected. For example:

Listing 933: show_range.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Range is
4
5     type Float_6_Digits_Normalized is
6         digits 6
7         range -1.0 .. 1.0;
8
9     type Float_9_Digits_Normalized is
10        digits 9
11        range -1.0 .. 1.0;
12
13    F6_N : Float_6_Digits_Normalized;
14    F9_N : Float_9_Digits_Normalized;
15 begin
16    F6_N := 0.123456123;
17
18    Put_Line ("F6_N = "
19              & F6_N'Image);
20
21    F9_N := Float_9_Digits_Normalized (F6_N);
22    -- Converting from
23    -- Float_6_Digits_Normalized
24    -- to
25    -- Float_9_Digits_Normalized
26
27    Put_Line ("F9_N = "
28              & F9_N'Image);
29
```

(continues on next page)

(continued from previous page)

```

30  Put_Line
31    ("Float_6_Digits_Normalized'Size  : "
32     & Float_6_Digits_Normalized'Size'Image
33     & " bits");
34
35  Put_Line
36    ("Float_9_Digits_Normalized'Size  : "
37     & Float_9_Digits_Normalized'Size'Image
38     & " bits");
39  end Show_Range;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪Point_Range
 MD5: a0b0f02786d023d1dfc3bd02c3e92736

Runtime output

```

F6_N = 1.23456E-01
F9_N = 1.23456120E-01
Float_6_Digits_Normalized'Size  : 32 bits
Float_9_Digits_Normalized'Size  : 64 bits

```

In this example, we assign the F6_N object of Float_6_Digits_Normalized type to the F9_N object of Float_9_Digits_Normalized type. Of course, if a range is specified, the value of an object cannot be outside of the type's range:

Listing 934: show_range.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Range is
4
5      type Float_6_Digits_Normalized is
6          digits 6
7          range -1.0 .. 1.0;
8
9      F6_N : Float_6_Digits_Normalized;
10  begin
11      F6_N := 0.123456123;
12
13      Put_Line ("F6_N = "
14               & F6_N'Image);
15
16      F6_N := F6_N * 10.0 - 0.5;
17
18      Put_Line ("F6_N = "
19               & F6_N'Image);
20
21      F6_N := F6_N + 1.0;
22      -- ERROR: result of this operation
23      --         is outside of the interval
24      --         [-1.0, 1.0].
25
26      Put_Line ("F6_N = "
27               & F6_N'Image);
28  end Show_Range;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
↳Point_Range
MD5: 2bfbfc0ed5d485db478ff046a228448a

Runtime output

```
F6_N = 1.23456E-01  
F6_N = 7.34561E-01
```

```
raised CONSTRAINT_ERROR : show_range.adb:21 range check failed
```

In this example, the assignment `F6_N := F6_N + 1.0` overflows because the resulting value is outside of the range of the `Float_6_Digits_Normalized` type. In contrast, the assignment `F6_N := F6_N * 10.0 - 0.5` doesn't raise an exception because the resulting value is inside the range — even though the intermediate value (1.23456) resulting from the `F6_N * 10.0` operation is outside the type's range.

Range of derived floating-point types

We can specify a range when deriving from floating-point types. In fact, it's possible to specify a range when the parent type doesn't have any range constraints, or specify a subrange when the parent type already has a range constraint. For example:

Listing 935: show_range.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Show_Range is  
4  
5   type Float_6_Digits is  
6     digits 6;  
7  
8   type Float_6_Digits_Normalized is new  
9     Float_6_Digits  
10    range -1.0 .. 1.0;  
11  
12   type Float_6_Digits_Normalized_Positive is new  
13     Float_6_Digits_Normalized  
14    range 0.0 .. 1.0;  
15  
16   F6_N : Float_6_Digits_Normalized;  
17   F6_NP : Float_6_Digits_Normalized_Positive;  
18 begin  
19   F6_N := 0.123456123;  
20  
21   Put_Line ("F6_N = "  
22     & F6_N'Image);  
23  
24   F6_NP :=  
25     Float_6_Digits_Normalized_Positive (F6_N);  
26  
27   Put_Line ("F6_NP = "  
28     & F6_NP'Image);  
29  
30   Put_Line  
31     ("Float_6_Digits_Normalized'Size :"  
32       & Float_6_Digits_Normalized'Size'Image  
33       & " bits");  
34 end Show_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↳Point_Derived_Range
 MD5: 684bcfcbadda9d04ef93260fa8745f99

Runtime output

```
F6_N = 1.23456E-01
F6_NP = 1.23456E-01
Float_6_Digits_Normalized'Size : 32 bits
```

In this example, we derive the type `Float_6_Digits_Normalized` from `Float_6_Digits` and specify the normalized range `-1.0 .. 1.0`. Similarly, we derive `Float_6_Digits_Normalized_Positive` from `Float_6_Digits_Normalized` and constrain its range to positive numbers `(0.0 .. 1.0)`.

As we know, extending the range when deriving from a type isn't possible for any scalar type, be it discrete or real. Therefore, as expected, it's not possible to increase the range in this case:

Listing 936: `show_extended_range.ads`

```
1 package Show_Extended_Range is
2
3   type Float_6_Digits is
4     digits 6;
5
6   type Float_6_Digits_Normalized is new
7     Float_6_Digits
8     range -1.0 .. 1.0;
9
10  type Float_6_Digits_Normalized_Ext is new
11    Float_6_Digits_Normalized
12    range -2.0 .. 2.0;
13
14 end Show_Extended_Range;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↳Point_Derived_Range_Increase
 MD5: 91baa77990ed1488767f2c840dd157cf

Build output

```
show_extended_range.ads:12:14: warning: static value out of range of type "Float_6_
↳Digits_Normalized" defined at line 6 [enabled by default]
show_extended_range.ads:12:14: warning: Constraint_Error will be raised at run_
↳time [enabled by default]
show_extended_range.ads:12:22: warning: static value out of range of type "Float_6_
↳Digits_Normalized" defined at line 6 [enabled by default]
show_extended_range.ads:12:22: warning: Constraint_Error will be raised at run_
↳time [enabled by default]
```

Compilation fails for this example because we're trying to extend the range from `-1.0 .. 1.0` to `-2.0 .. 2.0` when deriving from the `Float_6_Digits_Normalized` type.

Range of floating-point subtypes

We can also specify a range when declaring a subtype of a floating-point type. For example:

Listing 937: show_floating_point_subtype_ranges.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Floating_Point_Subtype_Ranges is
4
5     type Float_6_Digits is
6         digits 6;
7
8     subtype Float_6_Digits_Subtype is
9         Float_6_Digits;
10    -- Same range as Float_6_Digits
11
12    subtype Float_6_Digits_Normalized is
13        Float_6_Digits
14        range -1.0 .. 1.0;
15
16    F6_N : Float_6_Digits_Normalized;
17 begin
18     F6_N := 0.123456123;
19
20     Put_Line ("F6_N = "
21              & F6_N'Image);
22 end Show_Floating_Point_Subtype_Ranges;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪ Point_Subtype_Range
 MD5: 876377a32643742afeca3f780afb160

Runtime output

```
F6_N = 1.23456E-01
```

In this example, we declare the `Float_6_Digits_Normalized` type as a subtype of `Float_6_Digits` and specify the normalized range `-1.0 .. 1.0`.

In the case of the subtype `Float_6_Digits_Subtype`, however, we haven't specified any range. Therefore, as expected, the range of the `Float_6_Digits` type is used.

Range of base type

Because the base type of a floating-point type is only constrained by the range of the root floating-point type, its range doesn't necessarily match the range of a floating-point type T — this is especially the case when we're specifying a custom range. For example:

Listing 938: show_floating_point_base_range.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Floating_Point_Base_Range is
4
5     type Float_6D is
6         digits 6;
7
8     type Float_6D_Norm is
9         digits 6
10        range -1.0 .. 1.0;
11
12 begin

```

(continues on next page)

(continued from previous page)

```

13
14 Put_Line
15   ("Float_6D'First      = "
16    & Float_6D'First'Image);
17 Put_Line
18   ("Float_6D'Last       = "
19    & Float_6D'Last'Image);
20
21 Put_Line
22   ("-----");
23
24 Put_Line
25   ("Float_6D_Norm'First  = "
26    & Float_6D_Norm'First'Image);
27 Put_Line
28   ("Float_6D_Norm'Last   = "
29    & Float_6D_Norm'Last'Image);
30
31 Put_Line
32   ("Float_6D_Norm'Base'First = "
33    & Float_6D_Norm'Base'First'Image);
34 Put_Line
35   ("Float_6D_Norm'Base'Last  = "
36    & Float_6D_Norm'Base'Last'Image);
37
38 end Show_Floating_Point_Base_Range;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Floating_
 ↪ Point_Base_Range
 MD5: 490dc81e47ced1b74dce80cbc2125213

Runtime output

```

Float_6D'First      = -3.40282E+38
Float_6D'Last       =  3.40282E+38
-----
Float_6D_Norm'First  = -1.00000E+00
Float_6D_Norm'Last   =  1.00000E+00
Float_6D_Norm'Base'First = -3.40282E+38
Float_6D_Norm'Base'Last  =  3.40282E+38

```

In this example, we see that the range of the range-constrained type `Float_6D_Norm` is restricted to `-1.0 .. 1.0`. On a desktop PC, the range of its base type — as well as the range of the `Float_6D` type — is typically `-3.40282E+38 .. 3.40282E+38`.

System max. base digits and max. digits values

There are two values associated with the maximum decimal precision of floating-point types: `System.Max_Digits` and `System.Max_Base_Digits`. They are dependent on the compiler capabilities, as well as hardware limitations:

Listing 939: `show_system_max_digits.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with System;
3
4 procedure Show_System_Max_Digits is
5 begin

```

(continues on next page)

(continued from previous page)

```
6   Put_Line ("System.Max_Digits      :"  
7             & System.Max_Digits'Image  
8             & " digits");  
9   Put_Line ("System.Max_Base_Digits :"  
10            & System.Max_Base_Digits'Image  
11            & " digits");  
12 end Show_System_Max_Digits;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.System_Max_Digits
MD5: 0c52be59b42a31428db0b0549fbfffd5

Runtime output

```
System.Max_Digits      : 18 digits  
System.Max_Base_Digits : 18 digits
```

On a typical desktop PC, we might see that the maximum decimal precision is the same in both cases:

- System.Max_Digits: 18 digits
- System.Max_Base_Digits: 18 digits

Note that this might not be the case for certain embedded devices.

For floating-point type declarations without a range constraint, the maximum decimal precision must not be greater than System.Max_Digits:

Listing 940: show_max_floating_point.ads

```
1 with System;  
2  
3 package Show_Max_Floating_Point is  
4  
5     type Max_Float is  
6         digits System.Max_Digits;  
7  
8 end Show_Max_Floating_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Max_Float
MD5: 2b9977cfd4197f86350206d70ae57318

Here, we're declaring the Max_Float using the maximum precision possible on the target platform.

When a range constraint is included in floating-point type declarations, the maximum decimal precision must not be greater than System.Max_Base_Digits:

Listing 941: show_max_floating_point.ads

```
1 with System;  
2  
3 package Show_Max_Floating_Point is  
4  
5     type Max_Float_Normalized is  
6         digits System.Max_Base_Digits  
7         range -1.0 .. 1.0;
```

(continues on next page)

(continued from previous page)

```

8
9 end Show_Max_Floating_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Max_Float_
 ↳ Range
 MD5: abd4c4996ea12146fd05a7ffda9477ed

Here, we're declaring the range-constrained Max_Float_Normalized using the maximum precision possible on the target platform.

Illegal floating-point type declarations

If a floating-point type declaration isn't supported by the Ada compiler or the target platform, it is considered illegal and, therefore, compilation will fail for that declaration. For example:

Listing 942: show_max_floating_point.ads

```

1 with System;
2
3 package Show_Max_Floating_Point is
4
5     type Max_Float is
6         digits System.Max_Digits + 1;
7
8 end Show_Max_Floating_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Floating_Point_Types.Max_Float_
 ↳ Illegal
 MD5: a1645bc2d42f50c06a1aac1be15c966c

Build output

```
show_max_floating_point.ads:6:31: error: digits value out of range, maximum is 18
gprbuild: *** compilation phase failed
```

In this example, we're trying to declare the Max_Float type with a decimal precision greater than the maximum supported precision. Therefore, compilation fails for this example.

25.8.8 Fixed-point types

We already discussed *fixed-point types* (page 107) in the Introduction to Ada course. Roughly speaking, fixed-point types can be thought as a way to mimic operations that look like floating-point types, but use discrete numeric types *in the background*. This has a big advantage for the implementation of certain numeric algorithms, as developers can use operations that look familiar because they resemble the ones they use with floating-point types.

In other languages

In many programming languages such as C, there's no built-in support for fixed-point types. This forces developers that need fixed-point types to circumvent this absence with sometimes cumbersome alternative. They could, for example, use integer types and introduce additional operations to match fixed-point operations. Alternatively,

frameworks or non-portable, compiler-specific extensions might be used in some cases. In contrast, the fact that Ada has built-in support for fixed-point types means that using these types is both portable and doesn't require extra efforts to circumvent limitations — such as the ones that originate from using integer types to emulate fixed-point operations.

As mentioned in the Introduction to Ada course, fixed-point types are classified in decimal fixed-point types and ordinary (binary).

Decimal fixed-point types are based on powers of ten and have the following syntax:

```
type <type-name> is
  delta <delta-value> digits <digits-value>;
```

Decimal fixed-point types are useful, for example, in many financial applications, where round-off errors from arithmetic operations are considered unacceptable.

Ordinary fixed-point types are based on powers of two (in their hardware implementation) and have the following syntax:

```
type <type-name> is
  delta <delta-value>
  range <lower-bound> .. <upper-bound>;
```

Ordinary fixed-point types can be found for example in some implementations for digital signal processing.

In the next sections, we discuss further details about these specific types. Next in this section, we introduce the concept of *small* and *delta* of fixed-point types, which are common for both kinds of fixed-point types.

Small and delta

The *small* and the *delta* of a fixed-point type indicate the numeric precision of that type. Let's discuss these concepts and how they differ from each other.

The *delta* corresponds to the value used for the **delta** in the type definition. For example, if we declare `type T3_D3 is delta 10.0 ** (-3) digits D;` then the *delta* is equal to the 10.0^{-3} that we used in the type definition.

The *small* of a type T is the smallest positive value used in the machine representation of the type. In other words, while the *delta* is primarily a user-selected value that (ideally) fits the requirements of the implementation, the *small* indicates how that *delta* is represented on the target machine.

The *small* must be at least equal to or smaller than the *delta*. In many cases, however, the *small* of a type T is equal to the *delta* of that type. In addition, note that these values aren't necessarily small numbers — in fact, they could be quite large.

We can use the T'*Small* and T'*Delta* attributes to retrieve the actual values of the *small* and *delta* of a fixed-point type T. (We discuss more details about these attributes [in another chapter](#) (page 719).) For example:

Listing 943: show_fixed_small_delta.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 procedure Show_Fixed_Small_Delta is
4
5   type Ordinary_Fixed_Point is
6     delta 0.25
```

(continues on next page)

(continued from previous page)

```

7      range -2.0 .. 2.0;
8
9  begin
10     Put_Line ("Ordinary_Fixed_Point'Small: "
11              & Ordinary_Fixed_Point'Small'Image);
12     Put_Line ("Ordinary_Fixed_Point'Delta: "
13              & Ordinary_Fixed_Point'Delta'Image);
14     Put_Line ("Ordinary_Fixed_Point'Size: "
15              & Ordinary_Fixed_Point'Size'Image);
16 end Show_Fixed_Small_Delta;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Small_Delta
MD5: b9cda138853519fd9cb6781fb8b38552

Runtime output

```

Ordinary_Fixed_Point'Small:  2.5000000000000000E-01
Ordinary_Fixed_Point'Delta:  2.5000000000000000E-01
Ordinary_Fixed_Point'Size:   5

```

In this example, we see the values for the compiler-selected *small* and the *delta* of type `Ordinary_Fixed_Point`. (Both are 0.25.)

When we declare a fixed-point data type, we must specify the *delta*. In contrast, providing a *small* in the type declaration is optional for ordinary fixed-point data types, but forbidden for decimal fixed-point types.

By default, the compiler automatically selects the *small*: this value is a power of ten for decimal fixed-point types and a power of two for ordinary fixed-point types. Also, for ordinary fixed-point types, we can specify the *small* by using the `Small` aspect.

As we mentioned before, the selected value for the *small* always follows the rule that it must be smaller or equal to the *delta*. For example:

Listing 944: `show_fixed_small_delta.adb`

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  procedure Show_Fixed_Small_Delta is
4
5      type Ordinary_Fixed_Point is
6          delta 0.2
7          range -2.0 .. 2.0;
8
9  begin
10     Put_Line ("Ordinary_Fixed_Point'Small: "
11              & Ordinary_Fixed_Point'Small'Image);
12     Put_Line ("Ordinary_Fixed_Point'Delta: "
13              & Ordinary_Fixed_Point'Delta'Image);
14     Put_Line ("Ordinary_Fixed_Point'Size: "
15              & Ordinary_Fixed_Point'Size'Image);
16 end Show_Fixed_Small_Delta;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Fixed_Small_Delta
MD5: 415e035b6d07247cf4ae1e4f6bf0923c

Runtime output

```

Ordinary_Fixed_Point'Small:  1.2500000000000000E-01
Ordinary_Fixed_Point'Delta:  2.0000000000000000E-01
Ordinary_Fixed_Point'Size:   6

```

In this example, the *delta* that we specified for `Ordinary_Fixed_Point` is 0.2, while the compiler-selected *small* is 2.0^{-3} .

For further reading...

As we've mentioned, the *small* and the *delta* need not actually be small numbers. They can be arbitrarily large. For instance, they could be 1.0, or 1000.0. Consider the following example:

Listing 945: `fixed_point_defs.ads`

```

1 package Fixed_Point_Defs is
2   S    : constant := 32;
3   Exp  : constant := 128;
4   D    : constant := 2.0 ** (-S + Exp + 1);
5
6   type Fixed is delta D
7     range -1.0 * 2.0 ** Exp ..
8           1.0 * 2.0 ** Exp - D;
9
10  pragma Assert (Fixed'Size = S);
11 end Fixed_Point_Defs;

```

Listing 946: `show_fixed_type_info.adb`

```

1 with Fixed_Point_Defs; use Fixed_Point_Defs;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Show_Fixed_Type_Info is
5 begin
6   Put_Line ("Size : "
7             & Fixed'Size'Image);
8   Put_Line ("Small : "
9             & Fixed'Small'Image);
10  Put_Line ("Delta : "
11            & Fixed'Delta'Image);
12  Put_Line ("First : "
13            & Fixed'First'Image);
14  Put_Line ("Last : "
15            & Fixed'Last'Image);
16 end Show_Fixed_Type_Info;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Fixed_Point_Types.Large_Small_Attribute
 MD5: 89672950b355060d250e0f5d7e2d40cb

Runtime output

```

Size : 32
Small : 1.58456325028528675E+29
Delta : 1.58456325028528675E+29
First : -340282366920938463463374607431768211456.0
Last : 340282366762482138434845932244680310784.0

```

In this example, the *small* of the `Fixed` type is actually quite large: 1.58456325028528675^{29} . (Also, the first and the last values are

large: -340,282,366,920,938,463,463,374,607,431,768,211,456.0 and 340,282,366,762,482,138,434,845,932,244,680,310,784.0, or approximately -3.4028^{38} and 3.4028^{38} .)

In this case, if we assign 1 or 1,000 to a variable F of this type, the actual value stored in F is zero. Feel free to try this out!

25.8.9 Big Numbers

As we've seen before, we can define numeric types in Ada with a high degree of precision. However, these normal numeric types in Ada are limited to what the underlying hardware actually supports. For example, any signed integer type — whether defined by the language or the user — cannot have a range greater than that of `System.Min_Int .. System.Max_Int` because those constants reflect the actual hardware's signed integer types. In certain applications, that precision might not be enough, so we have to rely on [arbitrary-precision arithmetic](#)¹⁵⁵. These so-called "big numbers" are limited conceptually only by available memory, in contrast to the underlying hardware-defined numeric types.

Ada supports two categories of big numbers: big integers and big reals — both are specified in child packages of the `Ada.Numerics.Big_Numbers` package:

Category	Package
Big Integers	<code>Ada.Numerics.Big_Numbers.Big_Integers</code>
Big Reals	<code>Ada.Numerics.Big_Numbers.Big_Real</code>

In the Ada Reference Manual

- [Big Numbers](#)¹⁵⁶
- [Big Integers](#)¹⁵⁷
- [Big Reals](#)¹⁵⁸

Overview

Let's start with a simple declaration of big numbers:

Listing 947: `show_simple_big_numbers.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  with Ada.Numerics.Big_Numbers.Big_Reals;
7  use  Ada.Numerics.Big_Numbers.Big_Reals;
8
9  procedure Show_Simple_Big_Numbers is
10     BI : Big_Integer;
11     BR : Big_Real;
12 begin
```

(continues on next page)

¹⁵⁵ https://en.wikipedia.org/wiki/arbitrary-precision_arithmetic

¹⁵⁶ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-5.html>

¹⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-6.html>

¹⁵⁸ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-7.html>

(continued from previous page)

```

13  BI := 12345678901234567890;
14  BR := 2.0 ** 1234;
15
16  Put_Line ("BI: " & BI'Image);
17  Put_Line ("BR: " & BR'Image);
18
19  BI := BI + 1;
20  BR := BR + 1.0;
21
22  Put_Line ("BI: " & BI'Image);
23  Put_Line ("BR: " & BR'Image);
24  end Show_Simple_Big_Numbers;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers
MD5: b6a5e9ad170b09cbbabeb3ce06cc958c

Runtime output

```

BI: 12345678901234567890
BR: ↪
↪ 29581122460809862906004469571610359078633968713537299223955620705065735079623892426105383724837
↪ 000
BI: 12345678901234567891
BR: ↪
↪ 29581122460809862906004469571610359078633968713537299223955620705065735079623892426105383724837
↪ 000

```

In this example, we're declaring the big integer BI and the big real BR, and we're incrementing them by one.

Naturally, we're not limited to using the + operator (such as in this example). We can use the same operators on big numbers that we can use with normal numeric types. In fact, the common unary operators (+, -, **abs**) and binary operators (+, -, *, /, **, Min and Max) are available to us. For example:

Listing 948: show_simple_big_numbers_operators.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use Ada.Numerics.Big_Numbers.Big_Integers;
5
6  procedure Show_Simple_Big_Numbers_Operators is
7      BI : Big_Integer;
8  begin
9      BI := 12345678901234567890;
10
11     Put_Line ("BI: " & BI'Image);
12
13     BI := -BI + BI / 2;
14     BI := BI - BI * 2;
15
16     Put_Line ("BI: " & BI'Image);
17 end Show_Simple_Big_Numbers_Operators;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Numbers_
↪ Operators

(continues on next page)

(continued from previous page)

MD5: 198708787bfcd6e16ec4fba718706af6

Runtime output

```
BI: 12345678901234567890
BI: 6172839450617283945
```

In this example, we're applying the four basic operators (+, -, *, /) on big integers.

Factorial

A typical example is the [factorial](https://en.wikipedia.org/wiki/Factorial)¹⁵⁹: a sequence of the factorial of consecutive small numbers can quickly lead to big numbers. Let's take this implementation as an example:

Listing 949: factorial.ads

```
1 function Factorial (N : Integer)
2   return Long_Long_Integer;
```

Listing 950: factorial.adb

```
1 function Factorial (N : Integer)
2   return Long_Long_Integer is
3   Fact : Long_Long_Integer := 1;
4 begin
5   for I in 2 .. N loop
6     Fact := Fact * Long_Long_Integer (I);
7   end loop;
8
9   return Fact;
10 end Factorial;
```

Listing 951: show_factorial.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Factorial;
4
5 procedure Show_Factorial is
6 begin
7   for I in 1 .. 50 loop
8     Put_Line (I'Image & "! = "
9               & Factorial (I)'Image);
10  end loop;
11 end Show_Factorial;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Factorial_Integer
MD5: 9b20469533706ef025a03b506a07b920
```

Runtime output

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

(continues on next page)

¹⁵⁹ <https://en.wikipedia.org/wiki/Factorial>

(continued from previous page)

```

6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000

```

```
raised CONSTRAINT_ERROR : factorial.adb:6 overflow check failed
```

Here, we're using **Long_Long_Integer** for the computation and return type of the Factorial function. (We're using **Long_Long_Integer** because its range is probably the biggest possible on the machine, although that is not necessarily so.) The last number we're able to calculate before getting an exception is $20!$, which basically shows the limitation of standard integers for this kind of algorithm. If we use big integers instead, we can easily display all numbers up to $50!$ (and more!):

Listing 952: factorial.ads

```

1 with Ada.Numerics.Big_Numbers.Big_Integers;
2 use  Ada.Numerics.Big_Numbers.Big_Integers;
3
4 function Factorial (N : Integer)
5     return Big_Integer;

```

Listing 953: factorial.adb

```

1 function Factorial (N : Integer)
2     return Big_Integer is
3     Fact : Big_Integer := 1;
4 begin
5     for I in 2 .. N loop
6         Fact := Fact * To_Big_Integer (I);
7     end loop;
8
9     return Fact;
10 end Factorial;

```

Listing 954: show_big_number_factorial.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Factorial;
4
5 procedure Show_Big_Number_Factorial is
6 begin
7     for I in 1 .. 50 loop
8         Put_Line (I'Image & "! = "
9                 & Factorial (I)'Image);
10    end loop;
11 end Show_Big_Number_Factorial;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Factorial_Big_Numbers
 MD5: d1f6464a3232d574d01f7ac14b822731

Runtime output

```

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
26! = 403291461126605635584000000
27! = 10888869450418352160768000000
28! = 304888344611713860501504000000
29! = 8841761993739701954543616000000
30! = 265252859812191058636308480000000
31! = 8222838654177922817725562880000000
32! = 263130836933693530167218012160000000
33! = 8683317618811886495518194401280000000
34! = 295232799039604140847618609643520000000
35! = 10333147966386144929666651337523200000000
36! = 371993326789901217467999448150835200000000
37! = 13763753091226345046315979581580902400000000
38! = 523022617466601111760007224100074291200000000
39! = 20397882081197443358640281739902897356800000000
40! = 815915283247897734345611269596115894272000000000
41! = 33452526613163807108170062053440751665152000000000
42! = 1405006117752879898543142606244511569936384000000000
43! = 60415263063373835637355132068513997507264512000000000
44! = 2658271574788448768043625811014615890319638528000000000
45! = 119622220865480194561963161495657715064383733760000000000
46! = 5502622159812088949850305428800254892961651752960000000000
47! = 258623241511168180642964355153611979969197632389120000000000
48! = 12413915592536072670862289047373375038521486354677760000000000
49! = 608281864034267560872252163321295376887552831379210240000000000
50! = 304140932017133780436126081660647688443776415689605120000000000000

```

As we can see in this example, replacing the **Long_Long_Integer** type by the `Big_Integer` type fixes the problem (the runtime exception) that we had in the previous version. (Note that we're using the `To_Big_Integer` function to convert from **Integer** to `Big_Integer`; we discuss these conversions next.)

Note that there is a limit to the upper bounds for big integers. However, this limit isn't dependent on the hardware types — as it's the case for normal numeric types —, but rather

compiler specific. In other words, the compiler can decide how much memory it wants to use to represent big integers.

Conversions

Most probably, we want to mix big numbers and *standard* numbers (i.e. integer and real numbers) in our application. In this section, we talk about the conversion between big numbers and standard types.

Validity

The package specifications of big numbers include subtypes that *ensure* that a actual value of a big number is valid:

Type	Subtype for valid values
Big Integers	Valid_Big_Integer
Big Reals	Valid_Big_Real

These subtypes include a contract for this check. For example, this is the definition of the Valid_Big_Integer subtype:

```
subtype Valid_Big_Integer is Big_Integer
with Dynamic_Predicate =>
    Is_Valid (Valid_Big_Integer),
    Predicate_Failure =>
        (raise Program_Error);
```

Any operation on big numbers is actually performing this validity check (via a call to the Is_Valid function). For example, this is the addition operator for big integers:

```
function "+" (L, R : Valid_Big_Integer)
return Valid_Big_Integer;
```

As we can see, both the input values to the operator as well as the return value are expected to be valid — the Valid_Big_Integer subtype triggers this check, so to say. This approach ensures that an algorithm operating on big numbers won't be using invalid values.

Conversion functions

These are the most important functions to convert between big number and *standard* types:

Category	To big number	From big number
Big Integers	To_Big_Integer	- To_Integer (Integer) - From_Big_Integer (other integer types)
Big Reals	To_Big_Real (floating-point types or fixed-point types)	From_Big_Real
	- To_Big_Real (Valid_Big_Integer) - To_Real (Integer)	Numerator, Denominator (Integer)

In the following sections, we discuss these functions in more detail.

Big integer to integer

We use the `To_Big_Integer` and `To_Integer` functions to convert back and forth between `Big_Integer` and `Integer` types:

Listing 955: `show_simple_big_integer_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  procedure Show_Simple_Big_Integer_Conversion is
7      BI : Big_Integer;
8      I  : Integer := 10000;
9  begin
10     BI := To_Big_Integer (I);
11     Put_Line ("BI: " & BI'Image);
12
13     I := To_Integer (BI + 1);
14     Put_Line ("I: " & I'Image);
15 end Show_Simple_Big_Integer_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Simple_Big_Integer_Conversion
MD5: 83effc9da9835d92f4c49ed03d7ed84a

Runtime output

```

BI:  10000
I:   10001
```

In addition, we can use the generic `Signed_Conversions` and `Unsigned_Conversions` packages to convert between `Big_Integer` and any signed or unsigned integer types:

Listing 956: `show_arbitrary_big_integer_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  procedure Show_Arbitrary_Big_Integer_Conversion is
7
8      type Mod_32_Bit is mod 2 ** 32;
9
10     package Long_Long_Integer_Conversions is new
11         Signed_Conversions (Long_Long_Integer);
12     use Long_Long_Integer_Conversions;
13
14     package Mod_32_Bit_Conversions is new
15         Unsigned_Conversions (Mod_32_Bit);
16     use Mod_32_Bit_Conversions;
17
18     BI  : Big_Integer;
19     LLI : Long_Long_Integer := 10000;
20     U_32 : Mod_32_Bit      := 2 ** 32 + 1;
21
```

(continues on next page)

(continued from previous page)

```

22 begin
23   BI := To_Big_Integer (LLI);
24   Put_Line ("BI:  " & BI'Image);
25
26   LLI := From_Big_Integer (BI + 1);
27   Put_Line ("LLI:  " & LLI'Image);
28
29   BI := To_Big_Integer (U_32);
30   Put_Line ("BI:  " & BI'Image);
31
32   U_32 := From_Big_Integer (BI + 1);
33   Put_Line ("U_32: " & U_32'Image);
34
35 end Show_Arbitrary_Big_Integer_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Arbitrary_Big_Integer_Conversion
 MD5: a89b42ff012c8729770eefa2d2b1f6c1

Runtime output

```

BI:      10000
LLI:     10001
BI:       1
U_32:    2

```

In this examples, we declare the `Long_Long_Integer_Conversions` and the `Mod_32_Bit_Conversions` to be able to convert between big integers and the **Long_Long_Integer** and the `Mod_32_Bit` types, respectively.

Note that, when converting from big integer to integer, we used the `To_Integer` function, while, when using the instances of the generic packages, the function is named `From_Big_Integer`.

Big real to floating-point types

When converting between big real and floating-point types, we have to instantiate the generic `Float_Conversions` package:

Listing 957: `show_big_real_floating_point_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Reals;
4  use  Ada.Numerics.Big_Numbers.Big_Reals;
5
6  procedure Show_Big_Real_Floating_Point_Conversion
7  is
8      type D10 is digits 10;
9
10     package D10_Conversions is new
11         Float_Conversions (D10);
12     use D10_Conversions;
13
14     package Long_Float_Conversions is new
15         Float_Conversions (Long_Float);
16     use Long_Float_Conversions;
17

```

(continues on next page)

(continued from previous page)

```

18  BR : Big_Real;
19  LF : Long_Float := 2.0;
20  F10 : D10      := 1.999;
21
22  begin
23    BR := To_Big_Real (LF);
24    Put_Line ("BR: " & BR'Image);
25
26    LF := From_Big_Real (BR + 1.0);
27    Put_Line ("LF: " & LF'Image);
28
29    BR := To_Big_Real (F10);
30    Put_Line ("BR: " & BR'Image);
31
32    F10 := From_Big_Real (BR + 0.1);
33    Put_Line ("F10: " & F10'Image);
34
35  end Show_Big_Real_Floating_Point_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Floating_
 ↗Point_Conversion
 MD5: 4ccb570b964d11d215660f5929f2709c

Runtime output

```

BR:      2.000
LF:      3.000000000000000E+00
BR:      1.999
F10:     2.0990000000E+00

```

In this example, we declare the `D10_Conversions` and the `Long_Float_Conversions` to be able to convert between big reals and the custom floating-point type `D10` and the `Long_Float` type, respectively. To do that, we use the `To_Big_Real` and the `From_Big_Real` functions.

Big real to fixed-point types

When converting between big real and ordinary fixed-point types, we have to instantiate the generic `Fixed_Conversions` package:

Listing 958: `show_big_real_fixed_point_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Reals;
4  use  Ada.Numerics.Big_Numbers.Big_Reals;
5
6  procedure Show_Big_Real_Fixed_Point_Conversion
7  is
8    D : constant := 2.0 ** (-31);
9    type TQ31 is delta D range -1.0 .. 1.0 - D;
10
11    package TQ31_Conversions is new
12      Fixed_Conversions (TQ31);
13    use TQ31_Conversions;
14
15    BR : Big_Real;
16    FQ31 : TQ31 := 0.25;

```

(continues on next page)

(continued from previous page)

```

17
18 begin
19   BR := To_Big_Real (FQ31);
20   Put_Line ("BR: " & BR'Image);
21
22   FQ31 := From_Big_Real (BR * 2.0);
23   Put_Line ("FQ31: " & FQ31'Image);
24
25 end Show_Big_Real_Fixed_Point_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Fixed_Point_Conversion
 MD5: 49f03e130ec34842cbac7a728a280821

Runtime output

```

BR:      0.250
FQ31:    0.50000000000

```

In this example, we declare the `TQ31_Conversions` to be able to convert between big reals and the custom fixed-point type `TQ31` type. Again, we use the `To_Big_Real` and the `From_Big_Real` functions for the conversions.

Note that there's no direct way to convert between decimal fixed-point types and big real types. (Of course, you could perform this conversion indirectly by using a floating-point or an ordinary fixed-point type in between.)

Big reals to (big) integers

We can also convert between big reals and big integers (or standard integers):

Listing 959: `show_big_real_big_integer_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  with Ada.Numerics.Big_Numbers.Big_Reals;
7  use  Ada.Numerics.Big_Numbers.Big_Reals;
8
9  procedure Show_Big_Real_Big_Integer_Conversion
10 is
11   I  : Integer;
12   BI : Big_Integer;
13   BR : Big_Real;
14
15 begin
16   I := 12345;
17   BR := To_Real (I);
18   Put_Line ("BR (from I): " & BR'Image);
19
20   BI := 123456;
21   BR := To_Big_Real (BI);
22   Put_Line ("BR (from BI): " & BR'Image);
23
24 end Show_Big_Real_Big_Integer_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Big_Integer_Conversion
 MD5: 26bf2a4704ce98709587eedab3391119

Runtime output

```
BR (from I): 12345.000
BR (from BI): 123456.000
```

Here, we use the `To_Real` and the `To_Big_Real` and functions for the conversions.

String conversions

In addition to that, we can use string conversions:

Listing 960: show_big_number_string_conversion.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  with Ada.Numerics.Big_Numbers.Big_Reals;
7  use  Ada.Numerics.Big_Numbers.Big_Reals;
8
9  procedure Show_Big_Number_String_Conversion
10 is
11     BI : Big_Integer;
12     BR : Big_Real;
13 begin
14     BI := From_String ("12345678901234567890");
15     BR := From_String ("12345678901234567890.0");
16
17     Put_Line ("BI: "
18               & To_String (Arg  => BI,
19                             Width => 5,
20                             Base => 2));
21     Put_Line ("BR: "
22               & To_String (Arg  => BR,
23                             Fore  => 2,
24                             Aft   => 6,
25                             Exp   => 18));
26 end Show_Big_Number_String_Conversion;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Number_String_Conversion
 MD5: aal1f19af04b0b901a086ac86151693a7

Runtime output

```
BI: 2#1010101101010100101010011000110011101011000111110000101011010010#
BR: 12.345678E+18
```

In this example, we use the `From_String` to convert a string to a big number. Note that the `From_String` function is actually called when converting a literal — because of the corresponding aspect for user-defined literals in the definitions of the `Big_Integer` and the `Big_Real` types.

i For further reading...

Big numbers are implemented using *user-defined literals* (page 344), which we discussed previously. In fact, these are the corresponding type declarations:

```
-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Integers;

type Big_Integer is private
  with Integer_Literal => From_Universal_Image,
       Put_Image      => Put_Image;

function From_Universal_Image
  (Arg : String)
return Valid_Big_Integer
renames From_String;

-- Declaration from
-- Ada.Numerics.Big_Numbers.Big_Reals;

type Big_Real is private
  with Real_Literal => From_Universal_Image,
       Put_Image    => Put_Image;

function From_Universal_Image
  (Arg : String)
return Valid_Big_Real
renames From_String;
```

As we can see in these declarations, the `From_String` function renames the `From_Universal_Image` function, which is being used for the user-defined literals.

Also, we call the `To_String` function to get a string for the big numbers. Naturally, using the `To_String` function instead of the `Image` attribute — as we did in previous examples — allows us to customize the format of the string that we display in the user message.

Other features of big integers

Now, let's look at two additional features of big integers:

- the natural and positive subtypes, and
- other available operators and functions.

Big positive and natural subtypes

Similar to integer types, big integers have the `Big_Natural` and `Big_Positive` subtypes to indicate natural and positive numbers. However, in contrast to the **Natural** and **Positive** subtypes, the `Big_Natural` and `Big_Positive` subtypes are defined via predicates rather than the simple ranges of normal (ordinary) numeric types:

```
subtype Natural is
  Integer range 0 .. Integer'Last;

subtype Positive is
  Integer range 1 .. Integer'Last;

subtype Big_Natural is Big_Integer
  with Dynamic_Predicate =>
    (if Is_Valid (Big_Natural))
```

(continues on next page)

(continued from previous page)

```

        then Big_Natural >= 0),
    Predicate_Failure =>
        (raise Constraint_Error);

subtype Big_Positive is Big_Integer
with Dynamic_Predicate =>
    (if Is_Valid (Big_Positive)
     then Big_Positive > 0),
    Predicate_Failure =>
        (raise Constraint_Error);

```

Therefore, we cannot simply use attributes such as `Big_Natural'First`. However, we can use the subtypes to ensure that a big integer is in the expected (natural or positive) range:

Listing 961: show_big_positive_natural.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  procedure Show_Big_Positive_Natural is
7      BI, D, N : Big_Integer;
8  begin
9      D := 3;
10     N := 2;
11     BI := Big_Natural (D / Big_Positive (N));
12
13     Put_Line ("BI: " & BI'Image);
14 end Show_Big_Positive_Natural;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Positive_Natural
MD5: 844b41f001c9aed9cb99decb221d93fd

Runtime output

```
BI: 1
```

By using the `Big_Natural` and `Big_Positive` subtypes in the calculation above (in the assignment to `BI`), we ensure that we don't perform a division by zero, and that the result of the calculation is a natural number.

Other operators for big integers

We can use the `mod` and `rem` operators with big integers:

Listing 962: show_big_integer_rem_mod.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Numerics.Big_Numbers.Big_Integers;
4  use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6  procedure Show_Big_Integer_Rem_Mod is
7      BI : Big_Integer;
8  begin
9      BI := 145 mod (-4);
10     Put_Line ("BI (mod): " & BI'Image);

```

(continues on next page)

(continued from previous page)

```
11
12     BI := 145 rem (-4);
13     Put_Line ("BI (rem): " & BI'Image);
14 end Show_Big_Integer_Rem_Mod;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Integer_Rem_Mod
MD5: 7347b617c51a3782921d997b3cfd5d37

Runtime output

```
BI (mod): -5
BI (rem):  1
```

In this example, we use the **mod** and **rem** operators in the assignments to BI.

Moreover, there's a `Greatest_Common_Divisor` function for big integers which, as the name suggests, calculates the greatest common divisor of two big integer values:

Listing 963: `show_big_integer_greatest_common_divisor.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use  Ada.Numerics.Big_Numbers.Big_Integers;
5
6 procedure Show_Big_Integer_Greatest_Common_Divisor
7 is
8     BI : Big_Integer;
9 begin
10     BI := Greatest_Common_Divisor (145, 25);
11     Put_Line ("BI: " & BI'Image);
12
13 end Show_Big_Integer_Greatest_Common_Divisor;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Integer_Greatest_
Common_Divisor
MD5: 27e2f7b4cbe20ec979b672f3e7edfdb7

Runtime output

```
BI:  5
```

In this example, we retrieve the greatest common divisor of 145 and 25 (i.e.: 5).

Big real and quotients

An interesting feature of big reals is that they support quotients. In fact, we can simply assign $2/3$ to a big real variable. (Note that we're able to omit the decimal points, as we write $2/3$ instead of $2.0 / 3.0$.) For example:

Listing 964: `show_big_real_quotient_conversion.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Numerics.Big_Numbers.Big_Reals;
4 use  Ada.Numerics.Big_Numbers.Big_Reals;
```

(continues on next page)

(continued from previous page)

```

5
6 procedure Show_Big_Real_Quotient_Conversion
7 is
8     BR    : Big_Real;
9 begin
10    BR := 2 / 3;
11    -- Same as:
12    -- BR := From_Quotient_String ("2 / 3");
13
14    Put_Line ("BR:  " & BR'Image);
15
16    Put_Line ("Q:      "
17              & To_Quotient_String (BR));
18
19    Put_Line ("Q numerator:  "
20              & Numerator (BR)'Image);
21    Put_Line ("Q denominator:  "
22              & Denominator (BR)'Image);
23 end Show_Big_Real_Quotient_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Real_Quotient_Conversion
 MD5: 97d78457d3f6d5e1810e461c2c7cd172

Runtime output

```

BR:    0.666
Q:     2 / 3
Q numerator:    2
Q denominator:  3

```

In this example, we assign `2 / 3` to `BR` — we could have used the `From_Quotient_String` function as well. Also, we use the `To_Quotient_String` to get a string that represents the quotient. Finally, we use the `Numerator` and `Denominator` functions to retrieve the values, respectively, of the numerator and denominator of the quotient (as big integers) of the big real variable.

Range checks

Previously, we've talked about the `Big_Natural` and `Big_Positive` subtypes. In addition to those subtypes, we have the `In_Range` function for big numbers:

Listing 965: show_big_numbers_in_range.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use Ada.Numerics.Big_Numbers.Big_Integers;
5
6 with Ada.Numerics.Big_Numbers.Big_Reals;
7 use Ada.Numerics.Big_Numbers.Big_Reals;
8
9 procedure Show_Big_Numbers_In_Range is
10
11     BI : Big_Integer;
12     BR : Big_Real;
13
14     BI_From : constant Big_Integer := 0;

```

(continues on next page)

(continued from previous page)

```

15  BI_To    : constant Big_Integer := 1024;
16
17  BR_From  : constant Big_Real := 0.0;
18  BR_To    : constant Big_Real := 1024.0;
19
20  begin
21      BI := 1023;
22      BR := 1023.9;
23
24      if In_Range (BI, BI_From, BI_To) then
25          Put_Line ("BI ( "
26                  & BI'Image
27                  & ") is in the "
28                  & BI_From'Image
29                  & " .. "
30                  & BI_To'Image
31                  & " range");
32      end if;
33
34      if In_Range (BR, BR_From, BR_To) then
35          Put_Line ("BR ( "
36                  & BR'Image
37                  & ") is in the "
38                  & BR_From'Image
39                  & " .. "
40                  & BR_To'Image
41                  & " range");
42      end if;
43
44  end Show_Big_Numbers_In_Range;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numerics.Big_Numbers.Big_Numbers_In_Range
MD5: ded52ef7e9ef13a83264940ff9d8bcb3

Runtime output

```

BI ( 1023) is in the  0 .. 1024 range
BR (1023.900) is in the  0.000 .. 1024.000 range

```

In this example, we call the `In_Range` function to check whether the big integer number (BI) and the big real number (BR) are in the range between 0 and 1024.

25.9 Numeric Attributes

25.9.1 Attributes of Modular Types

In this section, we discuss two attributes of modular types: `Modulus` and `Mod`. We also discuss operations on modular types.

***i* In the Ada Reference Manual**

- 3.5.4 Integer Types¹⁶⁰

¹⁶⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-4.html>

Modulus Attribute

The Modulus attribute returns the modulus of the modular type as a universal integer value. Let's get the modulus of the 32-bit Modular type that we've declared in the Num_Types package of the previous chapter:

Listing 966: num_types.ads

```

1 package Num_Types is
2
3     type Signed_Integer is range 1 .. 1_000_000;
4     type Modular is mod 2**32;
5
6 end Num_Types;
```

Listing 967: show_modular.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types;   use Num_Types;
4
5 procedure Show_Modular is
6     Modulus_Value : constant := Modular'Modulus;
7 begin
8     Put_Line (Modulus_Value'Image);
9 end Show_Modular;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Modular_1
MD5: 93b8743474559f85c79c46bd71da8641

Runtime output

4294967296

When we run this example, we get 4,294,967,296 — which is equal to 2^{32} .

Mod Attribute

Note

This section was originally written by Robert A. Duff and published as [Gem #26: The Mod Attribute](#)¹⁶¹.

Operations on signed integers can overflow: if the result is outside the base range, Constraint_Error will be raised. In our previous example, we declared the Signed_Integer type:

```
type Signed_Integer is range 1 .. 1_000_000;
```

The base range of Signed_Integer is the range of Signed_Integer'Base, which is chosen by the compiler, but is likely to be something like $-2^{31} .. 2^{31} - 1$. (Note: we discussed the Base attribute [in this section](#) (page 644).)

Operations on modular integers use modular (wraparound) arithmetic. For example:

¹⁶¹ <https://www.adacore.com/gems/gem-26>

Listing 968: show_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types;   use Num_Types;
4
5 procedure Show_Modular is
6   X : Modular;
7 begin
8   X := 1;
9   Put_Line (X'Image);
10
11   X := -X;
12   Put_Line (X'Image);
13 end Show_Modular;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Modular_1
MD5: e9ac61d2e43585f002fe2b79544ef9d7

Runtime output

```
1
4294967295
```

Negating X gives -1, which wraps around to $2^{**32} - 1$, i.e. all-one-bits.

But what about a type conversion from signed to modular? Is that a signed operation (so it should overflow) or is it a modular operation (so it should wrap around)? The answer in Ada is the former — that is, if you try to convert, say, **Integer**'(-1) to Modular, you will get `Constraint_Error`:

Listing 969: show_modular.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types;   use Num_Types;
4
5 procedure Show_Modular is
6   I : Integer := -1;
7   X : Modular := 1;
8 begin
9   X := Modular (I); -- raises Constraint_Error
10   Put_Line (X'Image);
11 end Show_Modular;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Modular_1
MD5: e8e1a1924efcbe770c719c29547bb863

Build output

```
show_modular.adb:9:09: warning: value not in range of type "Modular" defined at
↳ num_types.ads:4 [enabled by default]
show_modular.adb:9:09: warning: Constraint_Error will be raised at run time
↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_modular.adb:9 range check failed
```

To solve this problem, we can use the **Mod** attribute:

Listing 970: show_modular.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Num_Types; use Num_Types;
4
5 procedure Show_Modular is
6   I : constant Integer := -1;
7   X : Modular := 1;
8 begin
9   X := Modular'Mod (I);
10  Put_Line (X'Image);
11 end Show_Modular;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Modular_1
MD5: 572a753de946b7578c5f1b6a795ede98

Runtime output

```
4294967295
```

The **Mod** attribute will correctly convert from any integer type to a given modular type, using wraparound semantics.

Historically

In older versions of Ada — such as Ada 95 —, the only way to do this conversion is to use `Unchecked_Conversion`, which is somewhat uncomfortable. Furthermore, if you're trying to convert to a generic formal modular type, how do you know what size of signed integer type to use? Note that `Unchecked_Conversion` might malfunction if the source and target types are of different sizes.

The **Mod** attribute was added to Ada 2005 to solve this problem. Also, we can now safely use this attribute in generics. For example:

Listing 971: mod_attribute.ads

```

1 generic
2   type Formal_Modular is mod <>;
3 package Mod_Attribute is
4   function F return Formal_Modular;
5 end Mod_Attribute;
```


Listing 972: mod_attribute.adb

```

1 package body Mod_Attribute is
2
3   A_Signed_Integer : Integer := -1;
4
5   function F return Formal_Modular is
6   begin
7     return Formal_Modular'Mod
8       (A_Signed_Integer);
9   end F;
10
11 end Mod_Attribute;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Mod_Attribute

MD5: b2f227b8d4f14cd36508bf33c403f751

In this example, F will return the all-ones bit pattern, for whatever modular type is passed to Formal_Modular.

Operations on modular types

Modular types are particularly useful for bit manipulation. For example, we can use the **and**, **or**, **xor** and **not** operators for modular types.

Also, we can perform bit-shifting by multiplying or dividing a modular object with a power of two. For example, if M is a variable of modular type, then `M := M * 2 ** 3`; shifts the bits to the left by three bits. Likewise, `M := M / 2 ** 3` shifts the bits to the right. Note that the compiler selects the appropriate shifting operator when translating these operations to machine code — no actual multiplication or division will be performed.

Let's see a simple implementation of the CRC-CCITT (0x1D0F) algorithm:

Listing 973: crc_defs.ads

```

1 package Crc_Defs is
2
3   type Byte is mod 2 ** 8;
4   type Crc is mod 2 ** 16;
5
6   type Byte_Array is
7     array (Positive range <>) of Byte;
8
9   function Crc_CCITT (A : Byte_Array)
10     return Crc;
11
12   procedure Display (Crc_A : Crc);
13
14   procedure Display (A : Byte_Array);
15
16 end Crc_Defs;
```

Listing 974: crc_defs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Crc_Defs is
4
```

(continues on next page)

(continued from previous page)

```

5  package Byte_IO is new Modular_IO (Byte);
6  package Crc_IO  is new Modular_IO (Crc);
7
8  function Crc_CCITT (A : Byte_Array)
9      return Crc
10 is
11     X      : Byte;
12     Crc_A : Crc := 16#1d0f#;
13 begin
14     for I in A'Range loop
15         X := Byte (Crc_A / 2 ** 8) xor A (I);
16         X := X xor (X / 2 ** 4);
17         declare
18             Crc_X : constant Crc := Crc (X);
19         begin
20             Crc_A := Crc_A * 2 ** 8 xor
21                     Crc_X * 2 ** 12 xor
22                     Crc_X * 2 ** 5 xor
23                     Crc_X;
24         end;
25     end loop;
26
27     return Crc_A;
28 end Crc_CCITT;
29
30 procedure Display (Crc_A : Crc) is
31 begin
32     Crc_IO.Put (Crc_A);
33     New_Line;
34 end Display;
35
36 procedure Display (A : Byte_Array) is
37 begin
38     for E of A loop
39         Byte_IO.Put (E);
40         Put (" ", " ");
41     end loop;
42     New_Line;
43 end Display;
44
45 begin
46     Byte_IO.Default_Width := 1;
47     Byte_IO.Default_Base  := 16;
48     Crc_IO.Default_Width  := 1;
49     Crc_IO.Default_Base   := 16;
50 end Crc_Defs;

```

Listing 975: show_crc.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Crc_Defs;    use Crc_Defs;
3
4  procedure Show_Crc is
5      AA      : constant Byte_Array :=
6              (16#0#, 16#20#, 16#30#);
7      Crc_A : Crc;
8  begin
9      Crc_A := Crc_CCITT (AA);
10
11     Put ("Input array: ");
12     Display (AA);

```

(continues on next page)

(continued from previous page)

```

13
14     Put ("CRC-CCITT: ");
15     Display (Crc_A);
16 end Show_Crc;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Modular_Types.Mod_Crc_
 ↪ CCITT_Ada
 MD5: 9c66abfadcce92231295cbccad087912

Runtime output

Input array: 16#0#, 16#20#, 16#30#,
 CRC-CCITT: 16#21B9#

In this example, the core of the algorithm is implemented in the `Crc_CCITT` function. There, we use bit shifting — for instance, `* 2 ** 8` and `/ 2 ** 8`, which shift left and right, respectively, by eight bits. We also use the `xor` operator.

25.9.2 Attributes of Floating-Point Types

In this section, we discuss various attributes related to floating-point types.

In the Ada Reference Manual

- 3.5.8 Operations of Floating Point Types¹⁶²
- A.5.3 Attributes of Floating Point Types¹⁶³

Representation-oriented attributes

In this section, we discuss attributes related to the representation of floating-point types.

Attribute: `Machine_Radix`

`Machine_Radix` is an attribute that returns the radix of the hardware representation of a type. For example:

Listing 976: `show_machine_radix.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Radix is
4 begin
5     Put_Line
6       ("Float'Machine_Radix:      "
7        & Float'Machine_Radix'Image);
8     Put_Line
9       ("Long_Float'Machine_Radix:  "
10        & Long_Float'Machine_Radix'Image);
11    Put_Line
12      ("Long_Long_Float'Machine_Radix: "
13       & Long_Long_Float'Machine_Radix'Image);
14 end Show_Machine_Radix;

```

¹⁶² <http://www.ada-auth.org/standards/22rm/html/RM-3-5-8.html>

¹⁶³ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-3.html>

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪ Machine_Radix
 MD5: 88680df680f1db4ff803912850370551

Runtime output

```
Float'Machine_Radix:      2
Long_Float'Machine_Radix: 2
Long_Long_Float'Machine_Radix: 2
```

Usually, this value is two, as the radix is based on a binary system.

Attributes: Machine_Mantissa

Machine_Mantissa is an attribute that returns the number of bits reserved for the mantissa of the floating-point type. For example:

Listing 977: show_machine_mantissa.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Mantissa is
4 begin
5   Put_Line
6     ("Float'Machine_Mantissa:      "
7      & Float'Machine_Mantissa'Image);
8   Put_Line
9     ("Long_Float'Machine_Mantissa:  "
10      & Long_Float'Machine_Mantissa'Image);
11   Put_Line
12     ("Long_Long_Float'Machine_Mantissa: "
13      & Long_Long_Float'Machine_Mantissa'Image);
14 end Show_Machine_Mantissa;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪ Machine_Mantissa
 MD5: da946a90a454c6e8f68cbff1ec54c7d3

Runtime output

```
Float'Machine_Mantissa:      24
Long_Float'Machine_Mantissa: 53
Long_Long_Float'Machine_Mantissa: 64
```

On a typical desktop PC, as indicated by Machine_Mantissa, we have 24 bits for the floating-point mantissa of the **Float** type.

Machine_Emin and Machine_Emax

The Machine_Emin and Machine_Emax attributes return the minimum and maximum value, respectively, of the machine exponent the floating-point type. Note that, in all cases, the returned value is a universal integer. For example:

Listing 978: show_machine_emin_emax.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 procedure Show_Machine_Emin_Emax is
4 begin
5     Put_Line
6       ("Float'Machine_Emin:           "
7        & Float'Machine_Emin'Image);
8     Put_Line
9       ("Float'Machine_Emax:           "
10        & Float'Machine_Emax'Image);
11    Put_Line
12      ("Long_Float'Machine_Emin:       "
13       & Long_Float'Machine_Emin'Image);
14    Put_Line
15      ("Long_Float'Machine_Emax:       "
16       & Long_Float'Machine_Emax'Image);
17    Put_Line
18      ("Long_Long_Float'Machine_Emin:   "
19       & Long_Long_Float'Machine_Emin'Image);
20    Put_Line
21      ("Long_Long_Float'Machine_Emax:   "
22       & Long_Long_Float'Machine_Emax'Image);
23 end Show_Machine_Emin_Emax;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪Machine_Emin_Emax
 MD5: 9766e06faaf1fc2ca48dd0bc0461b247

Runtime output

```

Float'Machine_Emin:           -125
Float'Machine_Emax:           128
Long_Float'Machine_Emin:      -1021
Long_Float'Machine_Emax:      1024
Long_Long_Float'Machine_Emin: -16381
Long_Long_Float'Machine_Emax: 16384

```

On a typical desktop PC, the value of `Float'Machine_Emin` and `Float'Machine_Emax` is -125 and 128, respectively.

To get the actual minimum and maximum value of the exponent for a specific type, we need to use the `Machine_Radix` attribute that we've seen previously. Let's calculate the minimum and maximum value of the exponent for the `Float` type on a typical PC:

- Value of minimum exponent: `Float'Machine_Radix ** Float'Machine_Emin`.
 - In our target platform, this is $2^{-125} = 2.35098870164457501594 \times 10^{-38}$.
- Value of maximum exponent: `Float'Machine_Radix ** Float'Machine_Emax`.
 - In our target platform, this is $2^{128} = 3.40282366920938463463 \times 10^{38}$.

Attribute: Digits

Digits is an attribute that returns the requested decimal precision of a floating-point sub-type. Let's see an example:

Listing 979: show_digits.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Digits is

```

(continues on next page)

(continued from previous page)

```

4 begin
5   Put_Line ("Float'Digits:      "
6             & Float'Digits'Image);
7   Put_Line ("Long_Float'Digits:  "
8             & Long_Float'Digits'Image);
9   Put_Line ("Long_Long_Float'Digits: "
10            & Long_Long_Float'Digits'Image);
11 end Show_Digits;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↳ Digits
 MD5: cd1c88054f7d54703760a852d08acb6d

Runtime output

```

Float'Digits:      6
Long_Float'Digits: 15
Long_Long_Float'Digits: 18

```

Here, the requested decimal precision of the **Float** type is six digits.

Note that we said that **Digits** is the *requested* level of precision, which is specified as part of declaring a floating point type. We can retrieve the actual decimal precision with `Base'Digits`. For example:

Listing 980: show_base_digits.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Base_Digits is
4   type Float_D3 is new Float digits 3;
5 begin
6   Put_Line ("Float_D3'Digits:      "
7             & Float_D3'Digits'Image);
8   Put_Line ("Float_D3'Base'Digits:  "
9             & Float_D3'Base'Digits'Image);
10 end Show_Base_Digits;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↳ Base_Digits
 MD5: a2deb352f93511ab2a39d41f0b3f9512

Runtime output

```

Float_D3'Digits:      3
Float_D3'Base'Digits: 6

```

The requested decimal precision of the `Float_D3` type is three digits, while the actual decimal precision is six digits (on a typical desktop PC).

Attributes: Denorm, Signed_Zeros, Machine_Rounds, Machine_Overflows

In this section, we discuss attributes that return **Boolean** values indicating whether a feature is available or not in the target architecture:

- Denorm is an attribute that indicates whether the target architecture uses **denormalized numbers**¹⁶⁴.
- Signed_Zeros is an attribute that indicates whether the type uses a sign for zero values, so it can represent both -0.0 and 0.0.
- Machine_Rounds is an attribute that indicates whether rounding-to-nearest is used, rather than some other choice (such as rounding-toward-zero).
- Machine_Overflows is an attribute that indicates whether a Constraint_Error exception is (or is not) guaranteed to be raised when an operation with that type produces an overflow or divide-by-zero.

Listing 981: show_boolean_attributes.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Boolean_Attributes is
4  begin
5      Put_Line
6          ("Float'Denorm:          "
7           & Float'Denorm'Image);
8      Put_Line
9          ("Long_Float'Denorm:      "
10          & Long_Float'Denorm'Image);
11     Put_Line
12         ("Long_Long_Float'Denorm: "
13          & Long_Long_Float'Denorm'Image);
14     Put_Line
15         ("Float'Signed_Zeros:      "
16          & Float'Signed_Zeros'Image);
17     Put_Line
18         ("Long_Float'Signed_Zeros:  "
19          & Long_Float'Signed_Zeros'Image);
20     Put_Line
21         ("Long_Long_Float'Signed_Zeros: "
22          & Long_Long_Float'Signed_Zeros'Image);
23     Put_Line
24         ("Float'Machine_Rounds:     "
25          & Float'Machine_Rounds'Image);
26     Put_Line
27         ("Long_Float'Machine_Rounds:  "
28          & Long_Float'Machine_Rounds'Image);
29     Put_Line
30         ("Long_Long_Float'Machine_Rounds: "
31          & Long_Long_Float'Machine_Rounds'Image);
32     Put_Line
33         ("Float'Machine_Overflows:    "
34          & Float'Machine_Overflows'Image);
35     Put_Line
36         ("Long_Float'Machine_Overflows:  "
37          & Long_Float'Machine_Overflows'Image);
38     Put_Line
39         ("Long_Long_Float'Machine_Overflows: "
40          & Long_Long_Float'Machine_Overflows'Image);
41 end Show_Boolean_Attributes;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
↳ Machine_Rounds_Overflows
MD5: b3f72c212cf00e697fe144a87eb72339

¹⁶⁴ https://en.wikipedia.org/wiki/Subnormal_number

Runtime output

```
Float'Denorm:      TRUE
Long_Float'Denorm: TRUE
Long_Long_Float'Denorm: TRUE
Float'Signed_Zeros: TRUE
Long_Float'Signed_Zeros: TRUE
Long_Long_Float'Signed_Zeros: TRUE
Float'Machine_Rounds: TRUE
Long_Float'Machine_Rounds: TRUE
Long_Long_Float'Machine_Rounds: TRUE
Float'Machine_Overflows: FALSE
Long_Float'Machine_Overflows: FALSE
Long_Long_Float'Machine_Overflows: FALSE
```

On a typical PC, we have the following information:

- Denorm is true (i.e. the architecture uses denormalized numbers);
- Signed_Zeros is true (i.e. the standard floating-point types use a sign for zero values);
- Machine_Rounds is true (i.e. rounding-to-nearest is used for floating-point types);
- Machine_Overflows is false (i.e. there's no guarantee that a `Constraint_Error` exception is raised when an operation with a floating-point type produces an overflow or divide-by-zero).

Primitive function attributes

In this section, we discuss attributes that we can use to manipulate floating-point values.

Attributes: Fraction, Exponent and Compose

The Exponent and Fraction attributes return "parts" of a floating-point value:

- Exponent returns the machine exponent, and
- Fraction returns the mantissa part.

Compose is used to return a floating-point value based on a fraction (the mantissa part) and the machine exponent.

Let's see some examples:

Listing 982: show_exponent_fraction_compose.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Exponent_Fraction_Compose is
4 begin
5   Put_Line
6     ("Float'Fraction (1.0):      "
7      & Float'Fraction (1.0)'Image);
8   Put_Line
9     ("Float'Fraction (0.25):     "
10      & Float'Fraction (0.25)'Image);
11   Put_Line
12     ("Float'Fraction (1.0e-25):  "
13      & Float'Fraction (1.0e-25)'Image);
14   Put_Line
15     ("Float'Exponent (1.0):      "
16      & Float'Exponent (1.0)'Image);
17   Put_Line
18     ("Float'Exponent (0.25):     "
```

(continues on next page)

(continued from previous page)

```

19     & Float'Exponent (0.25)'Image);
20   Put_Line
21     ("Float'Exponent (1.0e-25): "
22     & Float'Exponent (1.0e-25)'Image);
23   Put_Line
24     ("Float'Compose (5.00000e-01, 1): "
25     & Float'Compose (5.00000e-01, 1)'Image);
26   Put_Line
27     ("Float'Compose (5.00000e-01, -1): "
28     & Float'Compose (5.00000e-01, -1)'Image);
29   Put_Line
30     ("Float'Compose (9.67141E-01, -83): "
31     & Float'Compose (9.67141E-01, -83)'Image);
32 end Show_Exponent_Fraction_Compose;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↳ Exponent_Fraction
 MD5: d2e61b6b9a7a50861145f6b65e9fac39

Runtime output

```

Float'Fraction (1.0):      5.00000E-01
Float'Fraction (0.25):    5.00000E-01
Float'Fraction (1.0e-25): 9.67141E-01
Float'Exponent (1.0):      1
Float'Exponent (0.25):    -1
Float'Exponent (1.0e-25): -83
Float'Compose (5.00000e-01, 1): 1.00000E+00
Float'Compose (5.00000e-01, -1): 2.50000E-01
Float'Compose (9.67141E-01, -83): 1.00000E-25

```

To understand this code example, we have to take this formula into account:

$$\text{Value} = \text{Fraction} \times \text{Machine\_Radix}^{\text{Exponent}}$$

Considering that the value of `Float'Machine_Radix` on a typical PC is two, we see that the value 1.0 is composed by a fraction of 0.5 and a machine exponent of one. In other words:

$$0.5 \times 2^1 = 1.0$$

For the value 0.25, we get a fraction of 0.5 and a machine exponent of -1, which is the result of $0.5 \times 2^{-1} = 0.25$. We can use the `Compose` attribute to perform this calculation. For example, `Float'Compose (0.5, -1) = 0.25`.

Note that `Fraction` is always between 0.5 and 0.999999 (i.e. < 1.0), except for denormalized numbers, where it can be < 0.5 .

Attribute: Scaling

Scaling is an attribute that scales a floating-point value based on the machine radix and a machine exponent passed to the function. For example:

Listing 983: show_scaling.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Scaling is
4 begin
5   Put_Line ("Float'Scaling (0.25, 1): "
6           & Float'Scaling (0.25, 1)'Image);

```

(continues on next page)

(continued from previous page)

```

7   Put_Line ("Float'Scaling (0.25, 2): "
8           & Float'Scaling (0.25, 2)'Image);
9   Put_Line ("Float'Scaling (0.25, 3): "
10          & Float'Scaling (0.25, 3)'Image);
11 end Show_Scaling;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪ Scaling
 MD5: 9fa821d32911b74ee4b4fde3f3adafd8

Runtime output

```
Float'Scaling (0.25, 1):  5.00000E-01
Float'Scaling (0.25, 2):  1.00000E+00
Float'Scaling (0.25, 3):  2.00000E+00
```

The scaling is calculated with this formula:

$$\text{scaling} = \text{value} \times \text{Machine\_Radix}^{\text{machine exponent}}$$

For example, on a typical PC with a machine radix of two, `Float'Scaling (0.25, 3) = 2.0` corresponds to

$$0.25 \times 2^3 = 2.0$$

Round-up and round-down attributes

Floor and Ceiling are attributes that returned the rounded-down or rounded-up value, respectively, of a floating-point value. For example:

Listing 984: show_floor_ceiling.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Floor_Ceiling is
4 begin
5   Put_Line ("Float'Floor (0.25):  "
6           & Float'Floor (0.25)'Image);
7   Put_Line ("Float'Ceiling (0.25): "
8           & Float'Ceiling (0.25)'Image);
9 end Show_Floor_Ceiling;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪ Floor_Ceiling
 MD5: 1344d54ae86b9fd4831d5f078eb655d4

Runtime output

```
Float'Floor (0.25):  0.00000E+00
Float'Ceiling (0.25):  1.00000E+00
```

As we can see in this example, the rounded-down value (floor) of 0.25 is 0.0, while the rounded-up value (ceiling) of 0.25 is 1.0.

Round-to-nearest attributes

In this section, we discuss three attributes used for rounding: `Rounding`, `Unbiased_Rounding`, `Machine_Rounding`. In all cases, the rounding attributes return the nearest integer value (as a floating-point value). For example, the rounded value for 4.8 is 5.0 because 5 is the closest integer value.

Let's see a code example:

Listing 985: show_roundings.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Roundings is
4  begin
5      Put_Line
6          ("Float'Rounding (0.5): "
7           & Float'Rounding (0.5)'Image);
8      Put_Line
9          ("Float'Rounding (1.5): "
10         & Float'Rounding (1.5)'Image);
11     Put_Line
12         ("Float'Rounding (4.5): "
13          & Float'Rounding (4.5)'Image);
14     Put_Line
15         ("Float'Rounding (-4.5): "
16          & Float'Rounding (-4.5)'Image);
17     Put_Line
18         ("Float'Unbiased_Rounding (0.5): "
19          & Float'Unbiased_Rounding (0.5)'Image);
20     Put_Line
21         ("Float'Unbiased_Rounding (1.5): "
22          & Float'Unbiased_Rounding (1.5)'Image);
23     Put_Line
24         ("Float'Machine_Rounding (0.5): "
25          & Float'Machine_Rounding (0.5)'Image);
26     Put_Line
27         ("Float'Machine_Rounding (1.5): "
28          & Float'Machine_Rounding (1.5)'Image);
29 end Show_Roundings;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↳ Rounding
 MD5: 3f78165f092a163339cb9593ff15a50d

Runtime output

```

Float'Rounding (0.5):  1.00000E+00
Float'Rounding (1.5):  2.00000E+00
Float'Rounding (4.5):  5.00000E+00
Float'Rounding (-4.5): -5.00000E+00
Float'Unbiased_Rounding (0.5):  0.00000E+00
Float'Unbiased_Rounding (1.5):  2.00000E+00
Float'Machine_Rounding (0.5):  1.00000E+00
Float'Machine_Rounding (1.5):  2.00000E+00
```

The difference between these attributes is the way they handle the case when a value is exactly in between two integer values. For example, 4.5 could be rounded up to 5.0 or rounded down to 4.0. This is the way each rounding attribute works in this case:

- `Rounding` rounds away from zero. Positive floating-point values are rounded up, while

negative floating-point values are rounded down when the value is between two integer values. For example:

- 4.5 is rounded-up to 5.0, i.e. `Float'Rounding (4.5) = Float'Ceiling (4.5) = 5.0`.
- -4.5 is rounded-down to -5.0, i.e. `Float'Rounding (-4.5) = Float'Floor (-4.5) = -5.0`.
- `Unbiased_Rounding` rounds toward the even integer. For example,
 - `Float'Unbiased_Rounding (0.5) = 0.0` because zero is the closest even integer, while
 - `Float'Unbiased_Rounding (1.5) = 2.0` because two is the closest even integer.
- `Machine_Rounding` uses the most appropriate rounding instruction available on the target platform. While this rounding attribute can potentially have the best performance, its result may be non-portable. For example, whether the rounding of 4.5 becomes 4.0 or 5.0 depends on the target platform.
 - If an algorithm depends on a specific rounding behavior, it's best to avoid the `Machine_Rounding` attribute. On the other hand, if the rounding behavior won't have a significant impact on the results, we can safely use this attribute.

Attributes: Truncation, Remainder, Adjacent

The `Truncation` attribute returns the *truncated* value of a floating-point value, i.e. the value corresponding to the integer part of a number rounded toward zero. This corresponds to the number before the radix point. For example, the truncation of 1.55 is 1.0 because the integer part of 1.55 is 1.

The `Remainder` attribute returns the remainder part of a division. For example, `Float'Remainder (1.25, 0.5) = 0.25`. Let's briefly discuss the details of this operation. The result of the division $1.25 / 0.5$ is 2.5. Here, 1.25 is the dividend and 0.5 is the divisor. The quotient and remainder of this division are 2 and 0.25, respectively. (Here, the quotient is an integer number, and the remainder is the floating-point part that remains.)

Note that the relation between quotient and remainder is defined in such a way that we get the original dividend back when we use the formula: "quotient x divisor + remainder = dividend". For the previous example, this means $2 \times 0.5 + 0.25 = 1.25$.

The `Adjacent` attribute is the next machine value towards another value. For example, on a typical PC, the adjacent value of a small value — say, 1.0×10^{-83} — towards zero is $+0.0$, while the adjacent value of this small value towards 1.0 is another small, but greater value — in fact, it's 1.40130×10^{-45} . Note that the first parameter of the `Adjacent` attribute is the value we want to analyze and the second parameter is the `Towards` value.

Let's see a code example:

Listing 986: `show_truncation_remainder_adjacent.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Truncation_Remainder_Adjacent is
4 begin
5   Put_Line
6     ("Float'Truncation (1.55): "
7     & Float'Truncation (1.55)'Image);
8   Put_Line
9     ("Float'Truncation (-1.55): "
10    & Float'Truncation (-1.55)'Image);
11  Put_Line
12    ("Float'Remainder (1.25, 0.25): "
```

(continues on next page)

(continued from previous page)

```

13     & Float'Remainder (1.25, 0.25)'Image);
14 Put_Line
15     ("Float'Remainder (1.25, 0.5): "
16     & Float'Remainder (1.25, 0.5)'Image);
17 Put_Line
18     ("Float'Remainder (1.25, 1.0): "
19     & Float'Remainder (1.25, 1.0)'Image);
20 Put_Line
21     ("Float'Remainder (1.25, 2.0): "
22     & Float'Remainder (1.25, 2.0)'Image);
23 Put_Line
24     ("Float'Adjacent (1.0e-83, 0.0): "
25     & Float'Adjacent (1.0e-83, 0.0)'Image);
26 Put_Line
27     ("Float'Adjacent (1.0e-83, 1.0): "
28     & Float'Adjacent (1.0e-83, 1.0)'Image);
29 end Show_Truncation_Remainder_Adjacent;

```

Attributes: Copy_Sign and Leading_Part

Copy_Sign is an attribute that returns a value where the sign of the second floating-point argument is multiplied by the magnitude of the first floating-point argument. For example, `Float'Copy_Sign (1.0, -10.0)` is -1.0. Here, the sign of the second argument (-10.0) is multiplied by the magnitude of the first argument (1.0), so the result is -1.0.

Leading_Part is an attribute that returns the *approximated* version of the mantissa of a value based on the specified number of leading bits for the mantissa. Let's see some examples:

- `Float'Leading_Part (3.1416, 1)` is 2.0 because that's the value we can represent with one leading bit.
 - Note that `Float'Fraction (2.0) = 0.5` — which can be represented with one leading bit in the mantissa — and `Float'Exponent (2.0) = 2.`
- If we increase the number of leading bits of the mantissa to two — by writing `Float'Leading_Part (3.1416, 2)` —, we get 3.0 because that's the value we can represent with two leading bits.
- If we increase again the number of leading bits to five — `Float'Leading_Part (3.1416, 5)` —, we get 3.125.
 - Note that, in this case `Float'Fraction (3.125) = 0.78125` and `Float'Exponent (3.125) = 2.`
 - The binary mantissa is actually `2#110_0100_0000_0000_0000_0000#`, which can be represented with five leading bits as expected: `2#110_01#`.
 - * We can get the binary mantissa by calculating `Float'Fraction (3.125) * Float (Float'Machine_Radix) ** (Float'Machine_Mantissa - 1)` and converting the result to binary format. The -1 value in the formula corresponds to the sign bit.

Attention

In this explanation about the Leading_Part attribute, we're talking about leading bits. Strictly speaking, however, this is actually a simplification, and it's only correct if Machine_Radix is equal to two — which is the case for most machines. Therefore, in most cases, the explanation above is perfectly acceptable.

However, if `Machine_Radix` is *not* equal to two, we cannot use the term "bits" anymore, but rather digits of the `Machine_Radix`.

Let's see some examples:

Listing 987: `show_copy_sign_leading_part_machine.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Copy_Sign_Leading_Part_Machine is
4  begin
5      Put_Line
6          ("Float'Copy_Sign (1.0, -10.0): "
7           & Float'Copy_Sign (1.0, -10.0)'Image);
8      Put_Line
9          ("Float'Copy_Sign (-1.0, -10.0): "
10         & Float'Copy_Sign (-1.0, -10.0)'Image);
11     Put_Line
12         ("Float'Copy_Sign (1.0, 10.0): "
13          & Float'Copy_Sign (1.0, 10.0)'Image);
14     Put_Line
15         ("Float'Copy_Sign (1.0, -0.0): "
16          & Float'Copy_Sign (1.0, -0.0)'Image);
17     Put_Line
18         ("Float'Copy_Sign (1.0, 0.0): "
19          & Float'Copy_Sign (1.0, 0.0)'Image);
20     Put_Line
21         ("Float'Leading_Part (1.75, 1): "
22          & Float'Leading_Part (1.75, 1)'Image);
23     Put_Line
24         ("Float'Leading_Part (1.75, 2): "
25          & Float'Leading_Part (1.75, 2)'Image);
26     Put_Line
27         ("Float'Leading_Part (1.75, 3): "
28          & Float'Leading_Part (1.75, 3)'Image);
29 end Show_Copy_Sign_Leading_Part_Machine;

```

Attribute: Machine

Not every real number is directly representable as a floating-point value on a specific machine. For example, let's take a value such as 1.0×10^{15} (or 1,000,000,000,000,000):

Listing 988: `show_float_value.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Float_Value is
4      package F_IO is new
5          Ada.Text_IO.Float_IO (Float);
6
7      V : Float;
8  begin
9      F_IO.Default_Fore := 3;
10     F_IO.Default_Aft  := 1;
11     F_IO.Default_Exp   := 0;
12
13     V := 1.0E+15;
14     Put ("1.0E+15 = ");
15     F_IO.Put (Item => V);
16     New_Line;

```

(continues on next page)

(continued from previous page)

```

17
18 end Show_Float_Value;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
↳ Float_Value
MD5: a7f80f7584ebaf39f2d5f9564c9c7d64

```

Runtime output

```
1.0E+15 = 999999986991000.0
```

If we run this example on a typical PC, we see that the expected value `1_000_000_000_000_000.0` was displayed as `999_999_986_991_000.0`. This is because 1.0×10^{15} isn't directly representable on this machine, so it has to be modified to a value that is actually representable (on the machine).

This *automatic* modification we've just described is actually hidden, so to say, in the assignment. However, we can make it more visible by using the `Machine (X)` attribute, which returns a version of `X` that is representable on the target machine. The `Machine (X)` attribute rounds (or truncates) `X` to either one of the adjacent machine numbers for the specific floating-point type of `X`. (Of course, if the real value of `X` is directly representable on the target machine, no modification is performed.)

In fact, we could rewrite the `V := 1.0E+15` assignment of the code example as `V := Float'Machine (1.0E+15)`, as we're never assigning a real value directly to a floating-pointing variable — instead, we're first converting it to a version of the real value that is representable on the target machine. In this case, `999999986991000.0` is a representable version of the real value 1.0×10^{15} . Of course, writing `V := 1.0E+15` or `V := Float'Machine (1.0E+15)` doesn't make any difference to the actual value that is assigned to `V` (in the case of this specific target architecture), as the conversion to a representable value happens automatically during the assignment to `V`.

There are, however, instances where using the `Machine` attribute does make a difference in the result. For example, let's say we want to calculate the difference between the original real value in our example (1.0×10^{15}) and the actual value that is assigned to `V`. We can do this by using the `Machine` attribute in the calculation:

Listing 989: show_machine_attribute.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Machine_Attribute is
4   package F_IO is new
5     Ada.Text_IO.Float_IO (Float);
6
7   V : Float;
8 begin
9   F_IO.Default_Fore := 3;
10  F_IO.Default_Aft  := 1;
11  F_IO.Default_Exp  := 0;
12
13  Put_Line
14    ("Original value: 1_000_000_000_000_000.0");
15
16  V := 1.0E+15;
17  Put ("Machine value: ");
18  F_IO.Put (Item => V);
19  New_Line;

```

(continues on next page)

(continued from previous page)

```

20
21   V := 1.0E+15 - Float'Machine (1.0E+15);
22   Put ("Difference: ");
23   F_IO.Put (Item => V);
24   New_Line;
25
26 end Show_Machine_Attribute;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Floating_Point_Types.
 ↪ Machine_Attribute
 MD5: c2db2cca028dc5811068f9b7f1bc209d

Runtime output

```

Original value: 1_000_000_000_000_000.0
Machine value:  999999986991000.0
Difference:     13008896.0
```

When we run this example on a typical PC, we see that the difference is roughly 1.3009×10^7 . (Actually, the value that we might see is 1.3008896×10^7 , which is a version of 1.3009×10^7 that is representable on the target machine.)

When we write `1.0E+15 - Float'Machine (1.0E+15)`:

- the first value in the operation is the universal real value 1.0×10^{15} , while
- the second value in the operation is a version of the universal real value 1.0×10^{15} that is representable on the target machine.

This also means that, in the assignment to `V`, we're actually writing `V := Float'Machine (1.0E+15 - Float'Machine (1.0E+15))`, so that:

1. we first get the intermediate real value that represents the difference between these values; and then
2. we get a version of this intermediate real value that is representable on the target machine.

This is the reason why we see 1.3008896×10^7 instead of 1.3009×10^7 when we run this application.

25.9.3 Attributes of Fixed-Point types

In this section, we discuss various attributes and operations related to fixed-point types.

In the Ada Reference Manual

- 3.5.10 Operations of Fixed Point Types¹⁶⁵
- A.5.4 Attributes of Fixed Point Types¹⁶⁶

Attributes of ordinary and decimal fixed-point types

Attribute: Machine_Radix

`Machine_Radix` is an attribute that returns the radix of the hardware representation of a type. For example:

¹⁶⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-5-10.html>

¹⁶⁶ <http://www.ada-auth.org/standards/22rm/html/RM-A-5-4.html>

Listing 990: show_fixed_machine_radix.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Fixed_Machine_Radix is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8 begin
9   Put_Line ("T3_D3'Machine_Radix: "
10            & T3_D3'Machine_Radix'Image);
11   Put_Line ("TQ31'Machine_Radix: "
12            & TQ31'Machine_Radix'Image);
13 end Show_Fixed_Machine_Radix;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
 ↪Fixed_Machine_Radix
 MD5: a09d060a58f76550e948a8245ffb5fde

Runtime output

```

T3_D3'Machine_Radix:  2
TQ31'Machine_Radix:  2

```

Usually, this value is two, as the radix is based on a binary system.

Attribute: Machine_Rounds and Machine_Overflows

In this section, we discuss attributes that return **Boolean** values indicating whether a feature is available or not in the target architecture:

- **Machine_Rounds** is an attribute that indicates what happens when the result of a fixed-point operation is inexact:
 - T'**Machine_Rounds** = **True**: inexact result is rounded;
 - T'**Machine_Rounds** = **False**: inexact result is truncated.
- **Machine_Overflows** is an attribute that indicates whether a **Constraint_Error** is guaranteed to be raised when a fixed-point operation with that type produces an overflow or divide-by-zero.

Listing 991: show_boolean_attributes.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Boolean_Attributes is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8 begin
9   Put_Line ("T3_D3'Machine_Rounds:  "
10            & T3_D3'Machine_Rounds'Image);
11   Put_Line ("TQ31'Machine_Rounds:  "
12            & TQ31'Machine_Rounds'Image);
13   Put_Line ("T3_D3'Machine_Overflows: "
14            & T3_D3'Machine_Overflows'Image);
15   Put_Line ("TQ31'Machine_Overflows: "

```

(continues on next page)

(continued from previous page)

```

16         & TQ31'Machine_Overflows'Image);
17 end Show_Boolean_Attributes;

```

Attribute: Small and Delta

The `Small` and `Delta` attributes return numbers that indicate the numeric precision of a fixed-point type. In many cases, the `Small` of a type `T` is equal to the `Delta` of that type — i.e. `T'Small = T'Delta`. Let's discuss each attribute and how they distinguish from each other.

When we declare an ordinary fixed-point data type, we must specify the *delta*. Specifying the *small*, however, is optional:

- If the *small* isn't specified, it is automatically selected by the compiler. In this case, the actual value of the *small* is an implementation-defined power of two — always following the rule that says: `T'Small <= T'Delta`.
- If we want, however, to specify the *small*, we can do that by using the `Small` aspect. In this case, it doesn't need to be a power of two.

For decimal fixed-point types, we cannot specify the *small*. In this case, it's automatically selected by the compiler, and it's always equal to the *delta*.

Let's see an example:

Listing 992: fixed_small_delta.ads

```

1 package Fixed_Small_Delta is
2   D3 : constant := 10.0 ** (-3);
3
4   type T3_D3 is delta D3 digits 3;
5
6   type TD3 is delta D3 range -1.0 .. 1.0 - D3;
7
8   D31 : constant := 2.0 ** (-31);
9   D15 : constant := 2.0 ** (-15);
10
11  type TQ31 is delta D31 range -1.0 .. 1.0 - D31;
12
13  type TQ15 is delta D15 range -1.0 .. 1.0 - D15
14    with Small => D31;
15 end Fixed_Small_Delta;

```

Listing 993: show_fixed_small_delta.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Fixed_Small_Delta; use Fixed_Small_Delta;
4
5 procedure Show_Fixed_Small_Delta is
6 begin
7   Put_Line ("T3_D3'Small: "
8             & T3_D3'Small'Image);
9   Put_Line ("T3_D3'Delta: "
10            & T3_D3'Delta'Image);
11  Put_Line ("T3_D3'Size: "
12            & T3_D3'Size'Image);
13  Put_Line ("-----");
14
15  Put_Line ("TD3'Small: "
16            & TD3'Small'Image);

```

(continues on next page)

(continued from previous page)

```

17   Put_Line ("T3'Delta: "
18             & T3'Delta'Image);
19   Put_Line ("T3'Size: "
20             & T3'Size'Image);
21   Put_Line ("-----");
22
23   Put_Line ("TQ31'Small: "
24             & TQ31'Small'Image);
25   Put_Line ("TQ31'Delta: "
26             & TQ31'Delta'Image);
27   Put_Line ("TQ32'Size: "
28             & TQ31'Size'Image);
29   Put_Line ("-----");
30
31   Put_Line ("TQ15'Small: "
32             & TQ15'Small'Image);
33   Put_Line ("TQ15'Delta: "
34             & TQ15'Delta'Image);
35   Put_Line ("TQ15'Size: "
36             & TQ15'Size'Image);
37 end Show_Fixed_Small_Delta;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
 ↪ Fixed_Small_Delta
 MD5: 0e811c7c0b92f05483b0ac7c3489dc3d

Runtime output

```

T3_D3'Small:  1.0000000000000000E-03
T3_D3'Delta:  1.0000000000000000E-03
T3_D3'Size:   11
-----
TD3'Small:    9.7656250000000000E-04
TD3'Delta:    1.0000000000000000E-03
TD3'Size:     11
-----
TQ31'Small:   4.65661287307739258E-10
TQ31'Delta:   4.65661287307739258E-10
TQ32'Size:    32
-----
TQ15'Small:   4.65661287307739258E-10
TQ15'Delta:   3.0517578125000000E-05
TQ15'Size:    32

```

As we can see in the output of the code example, the **Delta** attribute returns the value we used for **delta** in the type definition of the T3_D3, TD3, TQ31 and TQ15 types.

The TD3 type is an ordinary fixed-point type with the the same delta as the decimal T3_D3 type. In this case, however, TD3'Small is not the same as the TD3'Delta. On a typical desktop PC, TD3'Small is 2^{-10} , while the delta is 10^{-3} . (Remember that, for ordinary fixed-point types, if we don't specify the *small*, it's automatically selected by the compiler as a power of two smaller than or equal to the *delta*.)

In the case of the TQ15 type, we're specifying the *small* by using the Small aspect. In this case, the underlying size of the TQ15 type is 32 bits, while the precision we get when operating with this type is 16 bits. Let's see a specific example for this type:

Listing 994: show_fixed_small_delta.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Fixed_Small_Delta; use Fixed_Small_Delta;
4
5 procedure Show_Fixed_Small_Delta is
6   V : TQ15;
7 begin
8   Put_Line ("V'Size: " & V'Size'Image);
9
10  V := TQ15'Small;
11  Put_Line ("V: " & V'Image);
12
13  V := TQ15'Delta;
14  Put_Line ("V: " & V'Image);
15 end Show_Fixed_Small_Delta;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
 ↪Fixed_Small_Delta
 MD5: f2a71db911913d6fbf5343671599c0ae

Runtime output

```

V'Size: 32
V:  0.00000
V:  0.00003

```

In the first assignment, we assign `TQ15'Small` (2^{-31}) to `V`. This value is smaller than the type's *delta* (2^{-15}). Even though `V'Size` is 32 bits, `V'Delta` indicates 16-bit precision, and `TQ15'Small` requires 32-bit precision to be represented correctly. As a result, `V` has a value of zero after this assignment.

In contrast, after the second assignment — where we assign `TQ15'Delta` (2^{-15}) to `V` — we see, as expected, that `V` has the same value as the *delta*.

Attributes: Fore and Aft

The `Fore` and `Aft` attributes indicate the number of characters or digits needed for displaying a value in decimal representation. To be more precise:

- The `Fore` attribute refers to the digits before the decimal point and it returns the number of digits plus one for the sign indicator (which is either `-` or `space`), and it's always at least two.
- The `Aft` attribute returns the number of decimal digits that is needed to represent the delta after the decimal point.

Let's see an example:

Listing 995: show_fixed_fore_aft.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Fixed_Fore_Aft is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5
6   D : constant := 2.0 ** (-31);
7   type TQ31 is delta D range -1.0 .. 1.0 - D;
8

```

(continues on next page)

(continued from previous page)

```

9   Dec : constant T3_D3 := -0.123;
10  Fix : constant TQ31 := -TQ31'Delta;
11  begin
12    Put_Line ("T3_D3'Fore: "
13              & T3_D3'Fore'Image);
14    Put_Line ("T3_D3'Aft: "
15              & T3_D3'Aft'Image);
16
17    Put_Line ("TQ31'Fore: "
18              & TQ31'Fore'Image);
19    Put_Line ("TQ31'Aft: "
20              & TQ31'Aft'Image);
21    Put_Line ("----");
22    Put_Line ("Dec: "
23              & Dec'Image);
24    Put_Line ("Fix: "
25              & Fix'Image);
26  end Show_Fixed_Fore_Aft;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
 ↪ Fixed_Fore_Aft
 MD5: d031f74d967a96dee1c6a83ff4bd14cf

Runtime output

```

T3_D3'Fore:  2
T3_D3'Aft:   3
TQ31'Fore:   2
TQ31'Aft:   10
----
Dec: -0.123
Fix: -0.0000000005

```

As we can see in the output of the Dec and Fix variables at the bottom, the value of Fore is two for both T3_D3 and TQ31. This value corresponds to the length of the string "-0" displayed in the output for these variables (the first two characters of "-0.123" and "-0.0000000005").

The value of Dec'Aft is three, which matches the number of digits after the decimal point in "-0.123". Similarly, the value of Fix'Aft is 10, which matches the number of digits after the decimal point in "-0.0000000005".

Attributes of decimal fixed-point types

The attributes presented in this subsection are only available for decimal fixed-point types.

Attribute: Digits

Digits is an attribute that returns the number of significant decimal digits of a decimal fixed-point subtype. This corresponds to the value that we use for the **digits** in the definition of a decimal fixed-point type.

Let's see an example:

Listing 996: show_decimal_digits.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```

3 procedure Show_Decimal_Digits is
4   type T3_D6 is delta 10.0 ** (-3) digits 6;
5   subtype T3_D2 is T3_D6 digits 2;
6 begin
7   Put_Line ("T3_D6'Digits: "
8             & T3_D6'Digits'Image);
9   Put_Line ("T3_D2'Digits: "
10            & T3_D2'Digits'Image);
11 end Show_Decimal_Digits;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
 ↳ Decimal_Digits
 MD5: d46e67bd0f8b369918e7ab9ab4413ae7

Runtime output

```

T3_D6'Digits:  6
T3_D2'Digits:  2

```

In this example, T3_D6'Digits is six, which matches the value that we used for **digits** in the type definition of T3_D6. The same logic applies for subtypes, as we can see in the value of T3_D2'Digits. Here, the value is two, which was used in the declaration of the T3_D2 subtype.

Attribute: Scale

According to the Ada Reference Manual, the Scale attribute "indicates the position of the point relative to the rightmost significant digits of values" of a decimal type. For example:

- If the value of Scale is two, then there are two decimal digits after the decimal point.
- If the value of Scale is negative, that implies that the **Delta** is a power of 10 greater than 1, and it would be the number of zero digits that every value would end in.

The Scale corresponds to the N used in the **delta 10.0 ** (-N)** expression of the type declaration. For example, if we write **delta 10.0 ** (-3)** in the declaration of a type T, then the value of T'Scale is three.

Let's look at this complete example:

Listing 997: show_decimal_scale.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Decimal_Scale is
4   type TM3_D6 is delta 10.0 ** 3 digits 6;
5   type T3_D6 is delta 10.0 ** (-3) digits 6;
6   type T9_D12 is delta 10.0 ** (-9) digits 12;
7 begin
8   Put_Line ("TM3_D6'Scale: "
9             & TM3_D6'Scale'Image);
10  Put_Line ("T3_D6'Scale: "
11            & T3_D6'Scale'Image);
12  Put_Line ("T9_D12'Scale: "
13            & T9_D12'Scale'Image);
14 end Show_Decimal_Scale;

```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
↪Decimal_Scale
MD5: 56a99848cf31a9c69fe6d91ead73375a

Runtime output

```
TM3_D6'Scale: -3  
T3_D6'Scale: 3  
T9_D12'Scale: 9
```

In this example, we get the following values for the scales:

- TM3_D6'Scale = -3,
- T3_D6'Scale = 3,
- T9_D12 = 9.

As you can see, the value of Scale is directly related to the *delta* of the corresponding type declaration.

Attribute: Round

The Round attribute rounds a value of any real type to the nearest value that is a multiple of the *delta* of the decimal fixed-point type, rounding away from zero if exactly between two such multiples.

For example, if we have a type T with three digits, and we use a value with 10 digits after the decimal point in a call to T'Round, the resulting value will have three digits after the decimal point.

Note that the X input of an S'Round (X) call is a universal real value, while the returned value is of S'Base type.

Let's look at this example:

Listing 998: show_decimal_round.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Show_Decimal_Round is  
4   type T3_D3 is delta 10.0 ** (-3) digits 3;  
5 begin  
6   Put_Line ("T3_D3'Round (0.2774): "  
7     & T3_D3'Round (0.2774)'Image);  
8   Put_Line ("T3_D3'Round (0.2777): "  
9     & T3_D3'Round (0.2777)'Image);  
10 end Show_Decimal_Round;
```

Code block metadata

Project: Courses.Advanced_Ada.Data_Types.Numeric_Attributes.Fixed_Point_Types.
↪Decimal_Round
MD5: 153d9dae52fee750da30dd9152a03c37

Runtime output

```
T3_D3'Round (0.2774): 0.277  
T3_D3'Round (0.2777): 0.278
```

Here, the T3_D3 has a precision of three digits. Therefore, to fit this precision, 0.2774 is rounded to 0.277, and 0.2777 is rounded to 0.278.

CONTROL FLOW

26.1 Expressions

26.1.1 Expressions: Definition

According to the Ada Reference Manual, an expression "is a formula that defines the computation or retrieval of a value." Also, when an expression is evaluated, the computed or retrieved value always has an associated type known at compile-time.

Even though the definition above is very simple, Ada expressions are actually very flexible — and they can also be very complex. In fact, if you read the [corresponding section](#)¹⁶⁷ of the Ada Reference Manual, you'll quickly discover that they include elements such as relations, membership choices, terms and primaries. Some of these are classic elements of expressions in programming languages, although some of their forms are unique to Ada. In this section, we present examples of just some of these elements. For a complete overview, please refer to the Reference Manual.

In the Ada Reference Manual

- [4.4 Expressions](#)¹⁶⁸

Relations and simple expressions

Expressions usually consist of relations, which in turn consist of simple expressions. (There are more details to this, but we'll keep it simple for the moment.) Let's see a code example with a few expressions, which we dissect into the corresponding grammatical elements — we're going to discuss them later:

Listing 999: show_expression_elements.adb

```
1 procedure Show_Expression_Elements is
2   type Mode is (Off, A, B, C, D);
3
4   pragma Unreferenced (B, C, D);
5
6   subtype Active_Mode is Mode
7     range Mode'Succ (Off) .. Mode'Last;
8
9   M1, M2 : Mode;
10  Dummy   : Boolean;
11 begin
12   M1 := A;
13
```

(continues on next page)

¹⁶⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

¹⁶⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

(continued from previous page)

```

14  Dummy :=
15      M1 in Active_Mode
16          and then M2 in Off | A;
17
18      -- ^^^^^^^^^^^^^^^^^^ relation
19
20      -- ^^^^^^^^^^^^^^^^^^ relation
21      -- ^^^^^^^^^^^^^^^^^^ expression
22
23
24  Dummy :=
25      M1 in Active_Mode;
26      -- ^^ name
27      -- ^^ primary
28      -- ^^ factor
29      -- ^^ term
30      -- ^^ simple expression
31
32      -- ^^^^^^^^^^^^^ membership choice
33      -- ^^^^^^^^^^^^^ membership choice list
34
35      -- ^^^^^^^^^^^^^ relation
36      -- ^^^^^^^^^^^^^ expression
37
38  Dummy :=
39      M2 in Off | A;
40      -- ^^ name
41      -- ^^ primary
42      -- ^^ factor
43      -- ^^ term
44      -- ^^ simple expression
45
46      -- ^^^ membership choice
47      --      ^ membership choice
48      --      ^^^^^ membership choice list
49
50      -- ^^^^^^^^^^^^^ relation
51      -- ^^^^^^^^^^^^^ expression
52
53  end Show_Expression_Elements;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.

↳ Expression_Elements

MD5: a22e6f2d2bc181ce77097a1de204eb62

Build output

show_expression_elements.adb:9:08: warning: variable "M2" is read but never
 ↳ assigned [-gnatvw]

In this code example, we see three expressions. As we mentioned earlier, every expression has a type; here, the type of each expression is **Boolean**.

The first expression (M1 **in** Active_Mode **and then** M2 **in** Off | A) consists of two relations: M1 **in** Active_Mode and M2 **in** Off | A. Let's discuss some of the details.

The M1 **in** Active_Mode relation consists of the simple expression M1 and the membership choice list Active_Mode. (Here, the **in** keyword is part of the relation definition.) Also, as we see in the comments of the source code, the simple expression M1 is, at the same time, a term, a factor, a primary and a name.

Let's briefly talk about this chain of syntactic elements for simple expressions. Very roughly said, this is how we can break up simple expressions:

- a simple expression consists of terms;
- a term consists of factors;
- a factor consists of primaries;
- a primary can be one of those:
 - a numeric literal;
 - `null`;
 - a string literal;
 - *an aggregate* (page 516);
 - a name;
 - an allocator (like `new Integer`);
 - *a parenthesized expression* (page 729);
 - *a conditional expression* (page 731);
 - *a quantified expression* (page 734);
 - *a declare expression* (page 738).

For further reading...

The definition of simple expressions we've just seen is very simplified. In actuality, these are the grammatical elements specified in the Ada Reference Manual:

```
simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
  numeric_literal | null | string_literal | aggregate
| name | allocator | (expression)
| (conditional_expression) | (quantified_expression)
| (declare_expression)
```

Later on in this chapter, we discuss *conditional expressions* (page 731), *quantified expressions* (page 734) and *declare expressions* (page 738) in more details.

In the relation M2 `in` `Off` | A from the code example, `Off` | A is a membership choice list, and `Off` and A are membership choices.

For further reading...

Relations can actually be much more complicated than the one we just saw. In fact, this is the definition from the Ada Reference Manual:

```
expression ::=
  relation {and relation}
| relation {and then relation}
| relation {or relation}
| relation {or else relation}
| relation {xor relation}
```

```
relation ::=
    simple_expression
    [relational_operator simple_expression]
| simple_expression [not] in
    membership_choice_list
| raise_expression
```

Again, for more details, please refer to the [section on expressions](#)¹⁶⁹ of the Ada Reference Manual.

In the Ada Reference Manual

- 4.4 Expressions¹⁷⁰
- 4.5.2 Relational Operators and Membership Tests¹⁷¹

Numeric expressions

The expressions we've seen so far had the **Boolean** type. Although much of the grammar described in the Manual exists exclusively for Boolean operations, we can also write numeric expressions such as the following one:

Listing 1000: show_numeric_expressions.adb

```
1 procedure Show_Numeric_Expressions is
2   C1 : constant Integer := 5;
3   Dummy : Integer;
4 begin
5   Dummy :=
6     -2 ** 4 + 3 * C1 ** 8;
7     --      ^ numeric literal
8     --      ^ primary
9     --      ^^ name
10    --      ^^ primary
11    --      ^^^^^ factor
12    --      ^ multiplying operator
13    --      ^ numeric literal
14    --      ^ primary
15    --      ^ factor
16    --      ^^^^^ term
17    --
18    --      ^ numeric literal
19    --      ^ primary
20    --      ^ numeric literal
21    --      ^ primary
22    --      ^^^^^ factor
23    --      ^^^^^ term
24    --      ^ binary adding operator
25    --      ^ unary adding operator
26    --
27    --      ^^^^^^^^^^^^^^^^^ simple expression
28    --      ^^^^^^^^^^^^^^^^^ expression
29 end Show_Numeric_Expressions;
```

¹⁶⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

¹⁷⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

¹⁷¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-2.html>

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.
 ↳ Numeric_Expressions
 MD5: a3c902c7aa5b0afe30ae220256c3306a

In this code example, the expression `- 2 ** 4 + 3 * C1 ** 8` consists of just a single simple expression. (Note that simple expressions do not have to be "simple".) This simple expression consists of two terms: `2 ** 4` and `3 * C1 ** 8`. While the `2 ** 4` term is also a single factor, the `3 * C1 ** 8` term consists of two factors: `3` and `C1 ** 8`. Both the `2 ** 4` and the `C1 ** 8` factors consists of two primaries each:

- the `2 ** 4` factor has the primaries `2` and `4`,
- the `C1 ** 8` factor has the primaries `C1` and `8`.

In the Ada Reference Manual

- [4.4 Expressions](#)¹⁷²

Other expressions

Expressions aren't limited to the **Boolean** type or to numeric types. Indeed, expressions can be of any type, and the definition of primaries we've seen earlier on already hints in this direction — as it includes elements such as allocators. Because expressions are very flexible, covering all possible variations and combinations in this section is out of scope. Again, please refer to the [section on expressions](#)¹⁷³ of the Ada Reference Manual for further details.

Parenthesized expression

An interesting aspect of primaries is that, by using parentheses, we can embed an expression inside another expression. As an example, let's discuss the following expression and its elements:

Listing 1001: show_parenthesized_expressions.adb

```

1  procedure Show_Parenthesized_Expressions is
2      C1 : constant Integer := 4;
3      C2 : constant Integer := 5;
4
5      Dummy : Integer;
6  begin
7      Dummy :=
8          (2 + C1) * C2;
9          --      ^^      name
10         --      ^^      primary
11         --      ^^      factor
12         --      ^^      term
13
14         --      ^      numeric literal
15         --      ^      primary
16         --      ^      factor
17         --      ^      term
18
19         --      ^      binary adding operator
20         --      ^^^^^^^ simple expression

```

(continues on next page)

¹⁷² <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

¹⁷³ <http://www.ada-auth.org/standards/22rm/html/RM-4-4.html>

(continued from previous page)

```

21  --
22  --  ^^^^^^^^      expression
23  --  ^^^^^^^^      primary
24  --  ^^^^^^^^      factor
25  --
26  --          ^^ factor
27  --  ^^^^^^^^^^^^ term
28  --
29  --  ^^^^^^^^^^^^ simple expression
30  --
31  --  ^^^^^^^^^^^^ expression
32  end Show_Parenthesized_Expressions;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.
 ↳ Parenthesized_Expressions
 MD5: 5871d2b0cd33e4f562b96381e0f0d293

In this example, we first start with the single expression $(2 + C1) * C2$, which is also a simple expression consisting of just one term, which consists of two factors: $(2 + C1)$ and $C2$. The $(2 + C1)$ factor is also a primary. Now, because of the parentheses, we identify that the primary $(2 + C1)$ is an expression that is embedded in another expression.

i Important

To be fair, the existence of parentheses in a primary could also indicate other kinds of expressions, such as conditional or quantified expressions. However, differentiating between them is straightforward, as we'll see later on in this chapter.

We then proceed to parse the $(2 + C1)$ expression, which consists of the terms 2 and $C1$. As we've seen in the comments of the code example, each of these terms consists of one factor, which consists of one primary. In the end, after parsing the primaries, we identify that 2 is a numeric literal and $C1$ is a name.

Note that the usage of parentheses might lead to situations where we have expressions in potentially unsuspected places. For example, consider the following code example:

Listing 1002: show_name_in_expression.adb

```

1  procedure Show_Name_In_Expression is
2      type Mode is (Off, A, B, C, D);
3
4      M1 : Mode;
5  begin
6      M1 := A;
7
8      case M1 is
9          when Off | D =>
10             null;
11          when A | B | C =>
12             M1 := D;
13      end case;
14
15  end Show_Name_In_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.Name_In_Expression
 MD5: ec8fcbc511e6a372da4f0ad99d2619a5

Here, the case statement expects a selecting expression. In this case, M1 is identified as a name — after being identified as a relation, a simple expression, a term, a factor and a primary.

However, if we replace **case** M1 **is** by **case** (M1) **is**, (M1) is identified as a parenthesized expression, not as a name! This parenthesized expression is first parsed and evaluated, which might have implications in case statements, as we'll see *in another chapter* (page 755).

Let's look at another example, this time with a subprogram call:

Listing 1003: increment_by_one.ads

```
1 procedure Increment_By_One (I : in out Integer);
```

Listing 1004: increment_by_one.adb

```
1 procedure Increment_By_One (I : in out Integer) is  
2 begin  
3   I := I + 1;  
4 end Increment_By_One;
```

Listing 1005: show_name_in_expression.adb

```
1 with Increment_By_One;  
2  
3 procedure Show_Name_In_Expression is  
4   V : Integer := 0;  
5 begin  
6   Increment_By_One ((V));  
7 end Show_Name_In_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Expressions_Definition.Name_In_Expression
 MD5: 4805df49dc702e5cb365252e58742dd2

Build output

```
show_name_in_expression.adb:6:23: error: actual for "I" must be a variable  
gprbuild: *** compilation phase failed
```

The Increment_By_One procedure from this example expects a variable as an actual parameter because the parameter mode is **in out**. However, the (V) in the call to the procedure is interpreted as an expression, so we end up providing a value — the result of the expression — as the actual parameter instead of the V variable. Naturally, this is a compilation error. (Of course, writing Increment_By_One (V) fixes the error.)

26.1.2 Conditional Expressions

As we've seen before, we can write simple expressions such as I = 0 or D.Valid. A conditional expression, as the name implies, is an expression that contains a condition. This might be an "if-expression" (in the **if** ... **then** ... **else** form) or a "case-expression" (in the **case** ... **is when** => form).

The Max function in the following code example is an expression function implemented with a conditional expression — an if-expression, to be more precise:

Listing 1006: expr_func.ads

```
1 package Expr_Func is
2
3     function Max (A, B : Integer) return Integer is
4         (if A >= B then A else B);
5
6 end Expr_Func;
```

Let's say we have a system with four states Off, On, Waiting, and Invalid. For this system, we want to implement a function named Toggled that returns the *toggled* value of a state S. If the current value of S is either Off or On, the function toggles from Off to On (or from On to Off). For other values, the state remains unchanged — i.e. the returned value is the same as the input value. This is the implementation using a conditional expression:

Listing 1007: expr_func.ads

```
1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State is
6         (if S = Off
7          then On
8          elsif S = On
9          then Off
10         else S);
11
12 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↳ Conditional_If_Expressions_1
MD5: 7a99711afec0b481557f9874dfbf4de

As you can see, if-expressions may contain an **elsif** branch (and therefore be more complicated).

The code above corresponds to this more verbose version:

Listing 1008: expr_func.ads

```
1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State;
6
7 end Expr_Func;
```

Listing 1009: expr_func.adb

```
1 package body Expr_Func is
2
3     function Toggled (S : State) return State is
4     begin
5         if S = Off then
6             return On;
```

(continues on next page)

(continued from previous page)

```

7     elsif S = On then
8         return Off;
9     else
10        return S;
11    end if;
12    end Toggled;
13
14 end Expr_Func;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↳ Conditional_If_Expressions_2
MD5: 9e6cdf53c9c934f37e5717e1d230615a

If we compare the if-block of this code example to the if-expression of the previous example, we notice that the if-expression is just a simplified version without the **return** keyword and the **end if**. In fact, converting an if-block to an if-expression is quite straightforward.

We could also replace the if-expression used in the Toggled function above with a case-expression. For example:

Listing 1010: expr_func.ads

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State is
6         (case S is
7             when Off      => On,
8             when On       => Off,
9             when others   => S);
10
11 end Expr_Func;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
↳ Conditional_Case_Expressions_1
MD5: 0dd3a86f0872d1e8c3a81f7a17c44bd5

Note that we use commas in case-expressions to separate the alternatives (the **when** expressions). The code above corresponds to this more verbose version:

Listing 1011: expr_func.ads

```

1 package Expr_Func is
2
3     type State is (Off, On, Waiting, Invalid);
4
5     function Toggled (S : State) return State;
6
7 end Expr_Func;

```

Listing 1012: expr_func.adb

```

1 package body Expr_Func is
2
3     function Toggled (S : State) return State is
4         begin

```

(continues on next page)

(continued from previous page)

```

5      case S is
6          when Off    => return On;
7          when On     => return Off;
8          when others => return S;
9      end case;
10     end Toggled;
11
12 end Expr_Func;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Conditional_Expressions.
 ↳ Conditional_Case_Expressions_2
 MD5: db6a0737e3931c83c31f53e4da3d8a2b

If we compare the case block of this code example to the case-expression of the previous example, we notice that the case-expression is just a simplified version of the case block without the **return** keyword and the **end case**;, and with alternatives separated by commas instead of semicolons.

i In the Ada Reference Manual

- 4.5.7 Conditional Expressions¹⁷⁴

26.1.3 Quantified Expressions

Quantified expressions are **for** expressions using a quantifier — which can be either **all** or **some** — and a predicate. This kind of expressions let us formalize statements such as:

- "all values of array A must be zero" into **for all I in A'Range => A (I) = 0**, and
- "at least one value of array A must be zero" into **for some I in A'Range => A (I) = 0**.

In the quantified expression **for all I in A'Range => A (I) = 0**, the quantifier is **all** and the predicate is **A (I) = 0**. In the second expression, the quantifier is **some**. The result of a quantified expression is always a Boolean value.

For example, we could use the quantified expressions above and implement these two functions:

- **Is_Zero**, which checks whether all components of an array A are zero, and
- **Has_Zero**, which checks whether array A has at least one component of the array A is zero.

This is the complete code:

Listing 1013: int_arrays.ads

```

1 package Int_Arrays is
2
3     type Integer_Arr is
4         array (Positive range <>) of Integer;
5
6     function Is_Zero (A : Integer_Arr)
7         return Boolean is
8         (for all I in A'Range => A (I) = 0);
9

```

(continues on next page)

¹⁷⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-7.html>

(continued from previous page)

```

10  function Has_Zero (A : Integer_Arr)
11      return Boolean is
12      (for some I in A'Range => A (I) = 0);
13
14  procedure Display_Array (A      : Integer_Arr;
15                          Name : String);
16
17  end Int_Arrays;

```

Listing 1014: int_arrays.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Int_Arrays is
4
5      procedure Display_Array (A      : Integer_Arr;
6                              Name : String) is
7      begin
8          Put (Name & ": ");
9          for E of A loop
10             Put (E'Image & " ");
11          end loop;
12          New_Line;
13      end Display_Array;
14
15  end Int_Arrays;

```

Listing 1015: test_int_arrays.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Int_Arrays; use Int_Arrays;
4
5  procedure Test_Int_Arrays is
6      A : Integer_Arr := (0, 0, 1);
7  begin
8      Display_Array (A, "A");
9      Put_Line ("Is_Zero: "
10               & Boolean'Image (Is_Zero (A)));
11      Put_Line ("Has_Zero: "
12               & Boolean'Image (Has_Zero (A)));
13
14      A := (0, 0, 0);
15
16      Display_Array (A, "A");
17      Put_Line ("Is_Zero: "
18               & Boolean'Image (Is_Zero (A)));
19      Put_Line ("Has_Zero: "
20               & Boolean'Image (Has_Zero (A)));
21  end Test_Int_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
 ↳ Quantified_Expression_1
 MD5: 4bbda8a3830272748500f797f23f76fc

Runtime output

```
A:  0  0  1
```

(continues on next page)

(continued from previous page)

```

Is_Zero: FALSE
Has_Zero: TRUE
A:  0  0  0
Is_Zero: TRUE
Has_Zero: TRUE

```

As you might have expected, we can rewrite a quantified expression as a loop in the **for I in A'Range loop if ... return ...** form. In the code below, we're implementing Is_Zero and Has_Zero using loops and conditions instead of quantified expressions:

Listing 1016: int_arrays.ads

```

1 package Int_Arrays is
2
3     type Integer_Arr is
4         array (Positive range <>) of Integer;
5
6     function Is_Zero (A : Integer_Arr)
7         return Boolean;
8
9     function Has_Zero (A : Integer_Arr)
10        return Boolean;
11
12    procedure Display_Array (A      : Integer_Arr;
13                            Name : String);
14
15 end Int_Arrays;

```

Listing 1017: int_arrays.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Int_Arrays is
4
5     function Is_Zero (A : Integer_Arr)
6         return Boolean is
7     begin
8         for I in A'Range loop
9             if A (I) /= 0 then
10                return False;
11            end if;
12        end loop;
13
14        return True;
15    end Is_Zero;
16
17    function Has_Zero (A : Integer_Arr)
18        return Boolean is
19    begin
20        for I in A'Range loop
21            if A (I) = 0 then
22                return True;
23            end if;
24        end loop;
25
26        return False;
27    end Has_Zero;
28
29    procedure Display_Array (A      : Integer_Arr;
30                            Name : String) is

```

(continues on next page)

(continued from previous page)

```

31  begin
32      Put (Name & ": ");
33      for E of A loop
34          Put (E'Image & " ");
35      end loop;
36      New_Line;
37  end Display_Array;
38
39  end Int_Arrays;

```

Listing 1018: test_int_arrays.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Int_Arrays; use Int_Arrays;
4
5  procedure Test_Int_Arrays is
6      A : Integer_Arr := (0, 0, 1);
7  begin
8      Display_Array (A, "A");
9      Put_Line ("Is_Zero: "
10               & Boolean'Image (Is_Zero (A)));
11      Put_Line ("Has_Zero: "
12               & Boolean'Image (Has_Zero (A)));
13
14      A := (0, 0, 0);
15
16      Display_Array (A, "A");
17      Put_Line ("Is_Zero: "
18               & Boolean'Image (Is_Zero (A)));
19      Put_Line ("Has_Zero: "
20               & Boolean'Image (Has_Zero (A)));
21  end Test_Int_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
 ↳Quantified_Expression_2
 MD5: a957a8fd60e1849248efela84eae6afa

Runtime output

```

A: 0 0 1
Is_Zero: FALSE
Has_Zero: TRUE
A: 0 0 0
Is_Zero: TRUE
Has_Zero: TRUE

```

So far, we've seen quantified expressions using indices — e.g. **for all** I **in** A'Range => We could avoid indices in quantified expressions by simply using the E **of** A form. In this case, we can just write **for all** E **of** A => Let's adapt the implementation of Is_Zero and Has_Zero using this form:

Listing 1019: int_arrays.ads

```

1  package Int_Arrays is
2
3      type Integer_Arr is
4          array (Positive range <>) of Integer;

```

(continues on next page)

(continued from previous page)

```

5
6   function Is_Zero (A : Integer_Arr)
7       return Boolean is
8       (for all E of A => E = 0);
9
10  function Has_Zero (A : Integer_Arr)
11      return Boolean is
12      (for some E of A => E = 0);
13
14  end Int_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Quantified_Expression.
 ↳ Quantified_Expression_3
 MD5: 059d12a6529483ebcc5db23dc6262896

Here, we're checking the components E of the array A and comparing them against zero.

i In the Ada Reference Manual

- 4.5.8 Quantified Expressions¹⁷⁵

26.1.4 Declare Expressions

So far, we've seen expressions that make use of existing objects declared outside of the expression. Sometimes, we might want to declare constant objects inside the expression, so we can use them locally in the expression. Similarly, we might want to rename an object and use the renamed object in an expression. In those cases, we can use a declare expression.

A declare expression allows for declaring or renaming objects within an expression:

Listing 1020: p.ads

```

1  package P is
2
3      function Max (A, B : Integer) return Integer is
4          (declare
5              Bigger_A : constant Boolean := (A >= B);
6              begin
7                  (if Bigger_A then A else B));
8
9  end P;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Simple_
 ↳ Declare_Expression
 MD5: c4773c3749eea045ac5db147fbac594b

The declare expression starts with the **declare** keyword and the usual object declarations, and it's followed by the **begin** keyword and the body. In this example, the body of the declare expression is a conditional expression.

Of course, the code above isn't really useful, so let's look at a more complete example:

¹⁷⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-8.html>

Listing 1021: integer_arrays.ads

```

1 package Integer_Arrays is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   function Sum (Arr : Integer_Array)
7     return Integer;
8
9   --
10  -- Expression function using
11  -- declare expression:
12  --
13  function Avg (Arr : Integer_Array)
14    return Float is
15    (declare
16      A : Integer_Array renames Arr;
17      S : constant Float := Float (Sum (A));
18      L : constant Float := Float (A'Length);
19    begin
20      S / L);
21
22 end Integer_Arrays;
```

Listing 1022: integer_arrays.adb

```

1 package body Integer_Arrays is
2
3   function Sum (Arr : Integer_Array)
4     return Integer is
5   begin
6     return Acc : Integer := 0 do
7       for V of Arr loop
8         Acc := Acc + V;
9       end loop;
10    end return;
11  end Sum;
12
13 end Integer_Arrays;
```

Listing 1023: show_integer_arrays.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2
3 with Integer_Arrays; use Integer_Arrays;
4
5 procedure Show_Integer_Arrays is
6   Arr : constant Integer_Array := [1, 2, 3];
7 begin
8   Put_Line ("Sum: "
9     & Sum (Arr)'Image);
10  Put_Line ("Avg: "
11    & Avg (Arr)'Image);
12 end Show_Integer_Arrays;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_Arrays

MD5: 30a035038508549822c819b60638133d

Runtime output

```
Sum: 6
Avg: 2.00000E+00
```

In this example, the Avg function is implemented using a declare expression. In this expression, A renames the Arr array, and S is a constant initialized with the value returned by the Sum function.

In the Ada Reference Manual

- 4.5.9 Declare Expressions¹⁷⁶

Restrictions in the declarative part

The declarative part of a declare expression is more restricted than the declarative part of a subprogram or declare block. In fact, we cannot:

- declare variables;
- declare constants of limited types;
- rename an object of limited type that is constructed within the declarative part;
- declare aliased constants;
- declare constants that make use of the **Access** or **Unchecked_Access** attributes in the initialization;
- declare constants of anonymous access type.

Let's see some examples of erroneous declarations:

Listing 1024: integer_arrays.ads

```
1 package Integer_Arrays is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     type Integer_Sum is limited private;
7
8     type Const_Integer_Access is
9         access constant Integer;
10
11     function Sum (Arr : Integer_Array)
12         return Integer;
13
14     function Sum (Arr : Integer_Array)
15         return Integer_Sum;
16
17     --
18     -- Expression function using
19     -- declare expression:
20     --
21     function Avg (Arr : Integer_Array)
22         return Float is
23         (declare
24             A : Integer_Array renames Arr;
25
26             S1 : aliased constant Integer := Sum (A);
```

(continues on next page)

¹⁷⁶ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-9.html>

(continued from previous page)

```

27      -- ERROR: aliased constant
28
29      S : Float := Float (S1);
30      L : Float := Float (A'Length);
31      -- ERROR: declaring variables
32
33      S2 : constant Integer_Sum := Sum (A);
34      -- ERROR: declaring constant of
35      --      limited type
36
37      A1 : Const_Integer_Access :=
38          S1'Unchecked_Access;
39      -- ERROR: using 'Unchecked_Access
40      --      attribute
41
42      A2 : access Integer := null;
43      -- ERROR: declaring object of
44      --      anonymous access type
45  begin
46      S / L);
47
48  private
49
50      type Integer_Sum is new Integer;
51
52  end Integer_Arrays;

```

Listing 1025: integer_arrays.adb

```

1  package body Integer_Arrays is
2
3      function Sum (Arr : Integer_Array)
4          return Integer is
5      begin
6          return Acc : Integer := 0 do
7              for V of Arr loop
8                  Acc := Acc + V;
9              end loop;
10         end return;
11     end Sum;
12
13     function Sum (Arr : Integer_Array)
14         return Integer_Sum is
15         (Integer_Sum (Integer'(Sum (Arr))));
16
17 end Integer_Arrays;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Declare_Expressions.Integer_
 Arrays_Error
 MD5: ea38f5067c849b85685d70ffc386f7a7

Build output

```

integer_arrays.ads:26:10: error: "aliased" not allowed in declare_expression
integer_arrays.ads:29:10: error: object renaming or constant declaration expected
integer_arrays.ads:30:10: error: object renaming or constant declaration expected
integer_arrays.ads:33:10: error: object renaming or constant declaration expected
integer_arrays.ads:38:19: error: "Unchecked_Access" attribute cannot occur in a_
  declare_expression

```

(continues on next page)

(continued from previous page)

```
integer_arrays.ads:42:15: error: anonymous access type not allowed in declare_
↳expression
gprbuild: *** compilation phase failed
```

In this version of the Avg function, we see many errors in the declarative part of the declare expression. If we convert the declare expression into an actual function implementation, however, those declarations won't trigger compilation errors. (Feel free to try this out!)

26.1.5 Reduction Expressions

Note

This feature was introduced in Ada 2022.

A reduction expression reduces a list of values into a single value. For example, we can reduce the list `[2, 3, 4]` to a single value:

- by adding the values of the list: $2 + 3 + 4 = 9$, or
- by multiplying the values of the list: $2 * 3 * 4 = 24$.

We write a reduction expression by using the Reduce attribute and providing the reducer and its initial value:

- the reducer is the operator (e.g.: `+` or `*`) that we use to *combine* the values of the list;
- the initial value is the value that we use before all other values of the list.

For example, if we use `+` as the operator and `0` as the initial value, we get the reduction expression: $0 + 2 + 3 + 4 = 9$. This can be implemented using an array:

Listing 1026: show_reduction_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   A : array (1 .. 3) of Integer;
5   I : Integer;
6 begin
7   A := [2, 3, 4];
8   I := A'Reduce ("+", 0);
9
10  Put_Line ("A = "
11           & A'Image);
12  Put_Line ("I = "
13           & I'Image);
14 end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↳Simple_Reduction_Expression
 MD5: 63c85aeff33e9ab3bf37bcb62559e0b2

Runtime output

```
A =
[ 2,  3,  4]
I = 9
```

Here, we have the array *A* with a list of values. The *A'Reduce* ("+", 0) expression reduces the list of values of *A* into a single value — in this case, an integer value that is stored in *I*. This statement is equivalent to:

```
I := 0;
for E of A loop
  I := I + E;
end loop;
```

Naturally, we can reduce the array using the * operator:

Listing 1027: show_reduction_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   A : array (1 .. 3) of Integer;
5   I : Integer;
6 begin
7   A := [2, 3, 4];
8   I := A'Reduce ("*", 1);
9
10  Put_Line ("A = "
11           & A'Image);
12  Put_Line ("I = "
13           & I'Image);
14 end Show_Reduction_Expression;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↳ Simple_Reduction_Expression
 MD5: 98e1de10863eed4bd12cc6ab1d7ce7ef

Runtime output

```
A =
[ 2,  3,  4]
I = 24
```

In this example, we call *A'Reduce* ("*", 1) to reduce the list. (Note that we use an initial value of one because it is the *identity element*¹⁷⁷ of a multiplication, so the complete operation is: $1 * 2 * 3 * 4 = 24$.)

In the Ada Reference Manual

- [Reduction Expressions](#)¹⁷⁸

Value sequences

In addition to arrays, we can apply reduction expression to value sequences, which consist of an iterated element association — for example, [*for* *I* *in* 1 .. 3 => *I* + 1]. We can simply *append* the reduction expression to a value sequence:

¹⁷⁷ https://en.wikipedia.org/wiki/Identity_element

¹⁷⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-5-10.html>

Listing 1028: show_reduction_expression.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   I : Integer;
5 begin
6   I := [for I in 1 .. 3 =>
7         I + 1]'Reduce ("+", 0);
8   Put_Line ("I = "
9             & I'Image);
10
11  I := [for I in 1 .. 3 =>
12        I + 1]'Reduce ("*", 1);
13  Put_Line ("I = "
14          & I'Image);
15 end Show_Reduction_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↪Reduction_Expression_Value_Sequences
 MD5: 25b75869e53aa3c8a8f8c821a05718c5

Runtime output

```

I = 9
I = 24

```

In this example, we create the value sequence `[for I in 1 .. 3 => I + 1]` and reduce it using the `+` and `*` operators. (Note that the operations in this example have the same results as in the previous examples using arrays.)

Custom reducers

In the previous examples, we've used standard operators such as `+` and `*` as the reducer. We can, however, write our own reducers and pass them to the `Reduce` attribute. For example:

Listing 1029: show_reduction_expression.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Reduction_Expression is
4   type Integer_Array is
5     array (Positive range <>) of Integer;
6
7   A : Integer_Array (1 .. 3);
8   I : Long_Integer;
9
10  procedure Accumulate
11    (Accumulator : in out Long_Integer;
12     Value       : Integer) is
13  begin
14    Accumulator := Accumulator
15                  + Long_Integer (Value);
16  end Accumulate;
17
18  begin
19    A := [2, 3, 4];
20    I := A'Reduce (Accumulate, 0);
21

```

(continues on next page)

(continued from previous page)

```

22   Put_Line ("A = "
23           & A'Image);
24   Put_Line ("I = "
25           & I'Image);
26 end Show_Reduction_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↳ Custom_Reducer_Procedure
 MD5: 1ed7cd1f3f5d5b8acda36b04afa955f0

Runtime output

```

A =
[ 2,  3,  4]
I = 9

```

In this example, we implement the `Accumulate` procedure as our reducer, which is called to accumulate the individual elements (integer values) of the list. We pass this procedure to the `Reduce` attribute in the `I := A'Reduce (Accumulate, 0)` statement, which is equivalent to:

```

I := 0;
for E of A loop
  Accumulate (I, E);
end loop;

```

A custom reducer must have the following parameters:

1. The accumulator parameter, which stores the interim result — and the final result as well, once all elements of the list have been processed.
2. The value parameter, which is a single element from the list.

Note that the accumulator type doesn't need to match the type of the individual components. In this example, we're using **Integer** as the component type, while the accumulator type is **Long_Integer**. (For this kind of reducers, using **Long_Integer** instead of **Integer** for the accumulator type makes lots of sense due to the risk of triggering overflows while the reducer is accumulating values — e.g. when accumulating a long list with larger numbers.)

In the example above, we've implemented the reducer as a procedure. However, we can also implement it as a function. In this case, the accumulated value is returned by the function:

Listing 1030: show_reduction_expression.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Reduction_Expression is
4      type Integer_Array is
5          array (Positive range <>) of Integer;
6
7      A : Integer_Array (1 .. 3);
8      I : Long_Integer;
9
10     function Accumulate
11         (Accumulator : Long_Integer;
12          Value       : Integer)
13     return Long_Integer is

```

(continues on next page)

(continued from previous page)

```

14   begin
15       return Accumulator + Long_Integer (Value);
16   end Accumulate;
17
18   begin
19       A := [2, 3, 4];
20       I := A'Reduce (Accumulate, 0);
21
22       Put_Line ("A = "
23                & A'Image);
24       Put_Line ("I = "
25                & I'Image);
26   end Show_Reduction_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
↳ Custom_Reducer_Function
MD5: 3bfc9b59e4667490e40921770990f52b

Runtime output

```

A =
[ 2,  3,  4]
I = 9

```

In this example, we converted the `Accumulate` procedure into a function (while the core implementation is essentially the same).

Note that the reduction expression remains the same, independently of whether we're using a procedure or a function as the reducer. Therefore, the statement with the reduction expression in this example is the same as in the previous example: `I := A'Reduce (Accumulate, 0);`. Now that we're using a function, this statement is equivalent to:

```

I := 0;
for E of A loop
    I := Accumulate (I, E);
end loop;

```

Other accumulator types

The accumulator type isn't restricted to scalars: in fact, we could use record types as well. For example:

Listing 1031: show_reduction_expression.adb

```

1   with Ada.Text_IO; use Ada.Text_IO;
2
3   procedure Show_Reduction_Expression is
4       type Integer_Array is
5           array (Positive range <>) of Integer;
6
7       A : Integer_Array (1 .. 3);
8
9       type Integer_Accumulator is record
10          Value : Long_Integer;
11          Count : Integer;
12       end record;
13
14       function Accumulate

```

(continues on next page)

(continued from previous page)

```

15     (Accumulator : Integer_Accumulator;
16      Value       : Integer)
17     return Integer_Accumulator is
18 begin
19     return (Value => Accumulator.Value
20            + Long_Integer (Value),
21            Count => Accumulator.Count + 1);
22 end Accumulate;
23
24 function Zero return Integer_Accumulator is
25 (Value => 0, Count => 0);
26
27 function Average (Acc : Integer_Accumulator)
28 return Float is
29 (Float (Acc.Value) / Float (Acc.Count));
30
31 Acc : Integer_Accumulator;
32
33 begin
34   A := [2, 3, 4];
35
36   Acc := A'Reduce (Accumulate, Zero);
37   Put_Line ("Acc = "
38            & Acc'Image);
39   Put_Line ("Avg = "
40            & Average (Acc)'Image);
41 end Show_Reduction_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↳ Reducer_Integer_Accumulator
 MD5: 95d61e18e7b719d0a25dc35cdbff6af2

Runtime output

```

Acc =
(VALUE => 9,
 COUNT => 3)
Avg = 3.00000E+00

```

In this example, we're using the `Integer_Accumulator` record type in our reducer — the `Accumulate` function. In this case, we're not only accumulating the values, but also counting the number of elements in the list. (Of course, we could have used `A'Length` for that as well.)

Also, we're not limited to numeric types: we can also create a reducer using strings as the accumulator type. In fact, we can display the initial value and the elements of the list by using unbounded strings:

Listing 1032: `show_reduction_expression.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Strings.Unbounded;
4 use Ada.Strings.Unbounded;
5
6 procedure Show_Reduction_Expression is
7   type Integer_Array is
8     array (Positive range <>) of Integer;
9

```

(continues on next page)

(continued from previous page)

```

10  A : Integer_Array (1 .. 3);
11
12  function Unbounded_String_List
13    (Accumulator : Unbounded_String;
14     Value       : Integer)
15    return Unbounded_String is
16  begin
17    return Accumulator
18           & ", " & Value'Image;
19  end Unbounded_String_List;
20
21  begin
22    A := [2, 3, 4];
23
24    Put_Line ("A = "
25             & A'Image);
26    Put_Line ("L = "
27             & To_String (A'Reduce
28                          (Unbounded_String_List,
29                           To_Unbounded_String ("0"))));
29  end Show_Reduction_Expression;
30

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Expressions.Reduction_Expressions.
 ↳ Reducer_String_Accumulator
 MD5: 557416f08f28a48110c0fa6909086629

Runtime output

```

A =
[ 2,  3,  4]
L = 0,  2,  3,  4

```

In this case, the "accumulator" is concatenating the initial value and individual values of the list into a string.

26.2 Statements

26.2.1 Simple and Compound Statements

We can classify statements as either simple or compound. Simple statements don't contain other statements; think of them as "atomic units" that cannot be further divided. Compound statements, on the other hand, may contain other — simple or compound — statements.

Here are some examples from each category:

Category	Examples
Simple statements	Null statement, assignment, subprogram call, etc.
Compound statements	If statement, case statement, loop statement, block statement

** In the Ada Reference Manual**

- 5.1 Simple and Compound Statements - Sequences of Statements¹⁷⁹

¹⁷⁹ <http://www.ada-auth.org/standards/22rm/html/RM-5-1.html>

26.2.2 Labels

We can use labels to identify statements in the code. They have the following format: `<<Some_Label>>`. We write them right before the statement we want to apply it to. Let's see an example of labels with simple statements:

Listing 1033: show_statement_identifier.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4   pragma Warnings (Off, "is not referenced");
5   begin
6     <<Show_Hello>> Put_Line ("Hello World!");
7     <<Show_Test>> Put_Line ("This is a test.");
8
9     <<Show_Separator>>
10    <<Show_Block_Separator>>
11    Put_Line ("=====");
12 end Show_Statement_Identifier;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Simple_Labels
MD5: 820f5963b476af5c04314fd4373d2286

Runtime output

```

Hello World!
This is a test.
=====
```

Here, we're labeling each statement. For example, we use the `Show_Hello` label to identify the `Put_Line ("Hello World!");` statement. Note that we can use multiple labels a single statement. In this code example, we use the `Show_Separator` and `Show_Block_Separator` labels for the same statement.

In the Ada Reference Manual

- 5.1 Simple and Compound Statements - Sequences of Statements¹⁸⁰

Labels and goto statements

Labels are mainly used in combination with `goto` statements. (Although pretty much uncommon, we could potentially use labels to indicate important statements in the code.) Let's see an example where we use a `goto` label; statement to *jump* to a specific label:

Listing 1034: show_cleanup.adb

```

1 procedure Show_Cleanup is
2   pragma Warnings (Off, "always false");
3
4   Some_Error : Boolean;
5   begin
6     Some_Error := False;
7
8     if Some_Error then
9       goto Cleanup;
```

(continues on next page)

¹⁸⁰ <http://www.ada-auth.org/standards/22rm/html/RM-5-1.html>

(continued from previous page)

```

10     end if;
11
12     <<Cleanup>> null;
13 end Show_Cleanup;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Goto
MD5: 0ce06582bbefae818d4da3b7d2d3436b

Here, we transfer the control to the *cleanup* statement as soon as an error is detected.

Use-case: Continue

Another use-case is that of a Continue label in a loop. Consider a loop where we want to skip further processing depending on a condition:

Listing 1035: show_continue.adb

```

1  procedure Show_Continue is
2      function Is_Further_Processing_Needed
3          (Dummy : Integer)
4          return Boolean
5      is
6      begin
7          -- Dummy implementation
8          return False;
9      end Is_Further_Processing_Needed;
10
11     A : constant array (1 .. 10) of Integer :=
12         (others => 0);
13 begin
14     for E of A loop
15
16         -- Some stuff here...
17
18         if Is_Further_Processing_Needed (E) then
19
20             -- Do more stuff...
21
22             null;
23         end if;
24     end loop;
25 end Show_Continue;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Continue_1
MD5: 115eeaf08d5fb072d707d6325fe9cfd0

In this example, we call the `Is_Further_Processing_Needed (E)` function to check whether further processing is needed or not. If it's needed, we continue processing in the `if` statement. We could simplify this code by just using a Continue label at the end of the loop and a `goto` statement:

Listing 1036: show_continue.adb

```

1  procedure Show_Continue is
2      function Is_Further_Processing_Needed
3          (Dummy : Integer)

```

(continues on next page)

(continued from previous page)

```

4      return Boolean
5  is
6  begin
7      -- Dummy implementation
8      return False;
9  end Is_Further_Processing_Needed;
10
11  A : constant array (1 .. 10) of Integer :=
12      (others => 0);
13  begin
14      for E of A loop
15
16          -- Some stuff here...
17
18          if not Is_Further_Processing_Needed (E) then
19              goto Continue;
20          end if;
21
22          -- Do more stuff...
23
24          <<Continue>>
25      end loop;
26  end Show_Continue;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Label_Continue_2
MD5: 260b52ead782adf76eee5cf3c4e8332b

Here, we use a Continue label at the end of the loop and jump to it in the case that no further processing is needed. Note that, in this example, we don't have a statement after the Continue label because the label itself is at the end of a statement — to be more specific, at the end of the loop statement. In such cases, there's an implicit **null** statement.

Historically

Since Ada 2012, we can simply write:

```

loop
    -- Some statements...

    <<Continue>>
end loop;

```

If a label is used at the end of a sequence of statements, a **null** statement is implied. In previous versions of Ada, however, that is not the case. Therefore, when using those versions of the language, we must write at least a **null** statement:

```

loop
    -- Some statements...

    <<Continue>> null;
end loop;

```

Labels and compound statements

We can use labels with compound statements as well. For example, we can label a **for** loop:

Listing 1037: show_statement_identifier.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4   pragma Warnings (Off, "is not referenced");
5
6   Arr : constant array (1 .. 5) of Integer :=
7     (1, 4, 6, 42, 49);
8   Found : Boolean := False;
9 begin
10   <<Find_42>> for E of Arr loop
11     if E = 42 then
12       Found := True;
13       exit;
14     end if;
15   end loop;
16
17   Put_Line ("Found: " & Found'Image);
18 end Show_Statement_Identifier;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Loop_Label
MD5: 5ca80b5a379ba0b08ccfaa4c6eab64d5

Runtime output

Found: TRUE

For further reading...

In addition to labels, loops and block statements allow us to use a statement identifier. In simple terms, instead of writing <<Some_Label>>, we write Some_Label :.

We could rewrite the previous code example using a loop statement identifier:

Listing 1038: show_statement_identifier.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Statement_Identifier is
4   Arr : constant array (1 .. 5) of Integer :=
5     (1, 4, 6, 42, 49);
6   Found : Boolean := False;
7 begin
8   Find_42 : for E of Arr loop
9     if E = 42 then
10      Found := True;
11      exit Find_42;
12    end if;
13  end loop Find_42;
14
15  Put_Line ("Found: " & Found'Image);
16 end Show_Statement_Identifier;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Labels.Loop_Statement_Identifier
MD5: e52cb5eea9427addf3cabe64dd73bc2d

Runtime output

```
Found: TRUE
```

Loop statement and block statement identifiers are generally preferred over labels. Later in this chapter, we discuss this topic in more detail.

26.2.3 Exit loop statement

We've introduced bare loops back in the *Introduction to Ada course* (page 14). In this section, we'll briefly discuss loop names and exit loop statements.

A bare loop has this form:

```
loop
  exit when Some_Condition;
end loop;
```

We can name a loop by using a loop statement identifier:

```
Loop_Name:
  loop
    exit Loop_Name when Some_Condition;
  end loop Loop_Name;
```

In this case, we have to use the loop's name after **end loop**. Also, having a name for a loop allows us to indicate which loop we're exiting from: **exit** Loop_Name **when**.

Let's see a complete example:

Listing 1039: show_vector_cursor_iteration.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Containers.Vectors;
3
4 procedure Show_Vector_Cursor_Iteration is
5
6   package Integer_Vectors is new
7     Ada.Containers.Vectors
8     (Index_Type => Positive,
9      Element_Type => Integer);
10
11   use Integer_Vectors;
12
13   V : constant Vector := 20 & 10 & 0 & 13;
14   C : Cursor;
15 begin
16   C := V.First;
17   Put_Line ("Vector elements are: ");
18
19   Show_Elements :
20     loop
21       exit Show_Elements when C = No_Element;
22
23       Put_Line ("Element: "
24                & Integer'Image (V (C)));
25       C := Next (C);
26     end loop Show_Elements;
27
28 end Show_Vector_Cursor_Iteration;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Exit_Loop_Statement.Exit_Named_Loop
MD5: b77353f6ed98f8ddb32c73c47d249020

Runtime output

Vector elements are:
Element: 20
Element: 10
Element: 0
Element: 13

Naming a loop is particularly useful when we have nested loops and we want to exit directly from the inner loop:

Listing 1040: show_inner_loop_exit.adb

```
1 procedure Show_Inner_Loop_Exit is
2   pragma Warnings (Off);
3
4   Cond : Boolean := True;
5 begin
6
7   Outer_Processing : loop
8
9     Inner_Processing : loop
10      exit Outer_Processing when Cond;
11    end loop Inner_Processing;
12
13  end loop Outer_Processing;
14
15 end Show_Inner_Loop_Exit;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Exit_Loop_Statement.Inner_Loop_Exit
MD5: b5c7434f1bf23c2cb8f81e4c13a31386

Here, we indicate that we exit from the Outer_Processing loop in case a condition Cond is met, even if we're actually within the inner loop.

In the Ada Reference Manual

- 5.7 Exit Statements¹⁸¹

26.2.4 If, case and loop statements

In the Introduction to Ada course, we talked about *if statements* (page 10), *loop statements* (page 13), and *case statements* (page 16). This is a very simple code example with these statements:

Listing 1041: show_if_case_loop_statements.adb

```
1 procedure Show_If_Case_Loop_Statements is
2   pragma Warnings (Off);
3
```

(continues on next page)

¹⁸¹ <http://www.ada-auth.org/standards/22rm/html/RM-5-7.html>

(continued from previous page)

```

4   Reset      : Boolean := False;
5   Increment  : Boolean := True;
6   Val        : Integer := 0;
7   begin
8       --
9       -- If statement
10      --
11      if Reset then
12          Val := 0;
13      elsif Increment then
14          Val := Val + 1;
15      else
16          Val := Val - 1;
17      end if;
18
19      --
20      -- Loop statement
21      --
22      for I in 1 .. 5 loop
23          Val := Val * 2 - I;
24      end loop;
25
26      --
27      -- Case statement
28      --
29      case Val is
30          when 0 .. 5 =>
31              null;
32          when others =>
33              Val := 5;
34      end case;
35
36  end Show_If_Case_Loop_Statements;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.
 ↳ Example
 MD5: 4fdc7f00e5218ed59d9eb050339567f1

In this section, we'll look into a more advanced detail about the case statement.

In the Ada Reference Manual

- 5.3 If Statements¹⁸²
- 5.4 Case Statements¹⁸³
- 5.5 Loop Statements¹⁸⁴

Case statements and expressions

As we know, the case statement has a choice expression (**case** Choice_Expression **is**), which is expected to be a discrete type. Also, this expression can be a function call or a type conversion, for example — in addition to being a variable or a constant.

As we discussed *earlier on* (page 729), if we use parentheses, the contents between those

¹⁸² <http://www.ada-auth.org/standards/22rm/html/RM-5-3.html>

¹⁸³ <http://www.ada-auth.org/standards/22rm/html/RM-5-4.html>

¹⁸⁴ <http://www.ada-auth.org/standards/22rm/html/RM-5-5.html>

parentheses is parsed as an expression. In the context of case statements, the expression is first evaluated before being used as a choice expression. Consider the following code example:

Listing 1042: scales.ads

```
1 package Scales is
2
3     type Satisfaction_Scale is (Very_Dissatisfied,
4                                 Dissatisfied,
5                                 OK,
6                                 Satisfied,
7                                 Very_Satisfied);
8
9     type Scale is range 0 .. 10;
10
11     function To_Satisfaction_Scale
12         (S : Scale)
13         return Satisfaction_Scale;
14
15 end Scales;
```

Listing 1043: scales.adb

```
1 package body Scales is
2
3     function To_Satisfaction_Scale
4         (S : Scale)
5         return Satisfaction_Scale
6     is
7         Satisfaction : Satisfaction_Scale;
8     begin
9         case (S) is
10             when 0 .. 2 =>
11                 Satisfaction := Very_Dissatisfied;
12             when 3 .. 4 =>
13                 Satisfaction := Dissatisfied;
14             when 5 .. 6 =>
15                 Satisfaction := OK;
16             when 7 .. 8 =>
17                 Satisfaction := Satisfied;
18             when 9 .. 10 =>
19                 Satisfaction := Very_Satisfied;
20         end case;
21
22         return Satisfaction;
23     end To_Satisfaction_Scale;
24
25 end Scales;
```

Listing 1044: show_case_statement_expression.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Scales;      use Scales;
4
5 procedure Show_Case_Statement_Expression is
6     Score : constant Scale := 0;
7 begin
8     Put_Line ("Score: "
9               & Scale'Image (Score)
10              & Satisfaction_Scale'Image (
```

(continues on next page)

(continued from previous page)

```

11         To_Satisfaction_Scale (Score)));
12
13 end Show_Case_Statement_Expression;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.Case_Statement_Expression
 MD5: 353ff771291e0c994ec052e818f9720c

Build output

```

scales.adb:9:07: error: missing case values: -128 .. -1
scales.adb:9:07: error: missing case values: 11 .. 127
gprbuild: *** compilation phase failed

```

When we try to compile this code example, the compiler complains about missing values in the `To_Satisfaction_Scale` function. As we mentioned in the [Introduction to Ada course](#) (page 16), every possible value for the choice expression needs to be covered by a unique branch of the case statement. In principle, it *seems* that we're actually covering all possible values of the `Scale` type, which ranges from 0 to 10. However, we've written `case (S) is` instead of `case S is`. Because of the parentheses, `(S)` is evaluated as an expression. In this case, the expected range of the case statement is not `Scale'Range`, but the range of its *base type* (page 644) `Scale'Base'Range`.

i In other languages

In C, the switch-case statement requires parentheses for the choice expression:

Listing 1045: main.c

```

1  #include <stdio.h>
2
3
4  int main(int argc, const char * argv[])
5  {
6      int s = 0;
7
8      switch (s)
9      {
10         case 0:
11         case 1:
12             printf("Value in the 0 -- 1 range\n");
13         default:
14             printf("Value > 1\n");
15     }
16 }

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.If_Case_Loop_Statements.Case_Statement_C
 MD5: 64ef6b15f1bdf14ca9273964ec5e1755

Runtime output

```

Value in the 0 -- 1 range
Value > 1

```

In Ada, parentheses aren't expected in the choice expression. Therefore, we shouldn't write `case (S) is` in a C-like fashion — unless, of course, we really want to evaluate an

expression in the case statement.

26.2.5 Block Statements

We've introduced block statements back in the *Introduction to Ada course* (page 19). They have this simple form:

Listing 1046: show_block_statement.adb

```
1 procedure Show_Block_Statement is
2   pragma Warnings (Off);
3   begin
4
5     -- BLOCK STARTS HERE:
6     declare
7       I : Integer;
8     begin
9       I := 0;
10      end;
11
12 end Show_Block_Statement;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Block_Statements.Simple_
↳Block_Statement
MD5: 61134b3899620c6d9ed68974fae33b5e

We can use an identifier when writing a block statement. (This is similar to loop statement identifiers that we discussed in the previous section.) In this example, we implement a block called `Simple_Block`:

Listing 1047: show_block_statement.adb

```
1 procedure Show_Block_Statement is
2   pragma Warnings (Off);
3   begin
4
5     Simple_Block : declare
6       I : Integer;
7     begin
8       I := 0;
9     end Simple_Block;
10
11 end Show_Block_Statement;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Block_Statements.Block_
↳Statement_Identifier
MD5: b327b7675931d9b994637671c806c7c3

Note that we must write `end Simple_Block;` when we use the `Simple_Block` identifier.

Block statement identifiers are useful:

- to indicate the begin and the end of a block — as some blocks might be long or nested in other blocks;
- to indicate the purpose of the block (i.e. as code documentation).

i In the Ada Reference Manual

- 5.6 Block Statements¹⁸⁵

26.2.6 Extended return statement

A common idiom in Ada is to build up a function result in a local object, and then return that object:

Listing 1048: show_return.adb

```

1  procedure Show_Return is
2
3      type Array_Of_Natural is
4          array (Positive range <>) of Natural;
5
6      function Sum (A : Array_Of_Natural)
7          return Natural
8      is
9          Result : Natural := 0;
10     begin
11         for Index in A'Range loop
12             Result := Result + A (Index);
13         end loop;
14         return Result;
15     end Sum;
16
17 begin
18     null;
19 end Show_Return;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
 ↪ Simple_Return
 MD5: 16e85a8cba869802f912627c40a64c20

Since Ada 2005, a notation called the extended return statement is available: this allows you to declare the result object and return it as part of one statement. It looks like this:

Listing 1049: show_extended_return.adb

```

1  procedure Show_Extended_Return is
2
3      type Array_Of_Natural is
4          array (Positive range <>) of Natural;
5
6      function Sum (A : Array_Of_Natural)
7          return Natural
8      is
9      begin
10         return Result : Natural := 0 do
11             for Index in A'Range loop
12                 Result := Result + A (Index);
13             end loop;
14         end return;
15     end Sum;
16
17 begin
```

(continues on next page)

¹⁸⁵ <http://www.ada-auth.org/standards/22rm/html/RM-5-6.html>

(continued from previous page)

```

18     null;
19 end Show_Extended_Return;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↳ Extended_Return
MD5: d6d6edaf800a0e346ff8ede13cbbbe100

The return statement here creates `Result`, initializes it to `0`, and executes the code between `do` and `end return`. When `end return` is reached, `Result` is automatically returned as the function result.

In the Ada Reference Manual

- 6.5 Return Statements¹⁸⁶

Other usages of extended return statements

Note

This section was originally written by Robert A. Duff and published as [Gem #10: Limited Types in Ada 2005](#)¹⁸⁷.

While the `extended_return_statement` was added to the language specifically to support *limited constructor functions* (page 1107), it comes in handy whenever you want a local name for the function result:

Listing 1050: `show_string_construct.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_String_Construct is
4
5      function Make_String
6          (S      : String;
7           Prefix : String;
8           Use_Prefix : Boolean) return String
9      is
10         Length : Natural := S'Length;
11     begin
12         if Use_Prefix then
13             Length := Length + Prefix'Length;
14         end if;
15
16         return Result : String (1 .. Length) do
17
18             -- fill in the characters
19             if Use_Prefix then
20                 Result
21                     (1 .. Prefix'Length) := Prefix;
22
23                 Result
24                     (Prefix'Length + 1 .. Length) := S;

```

(continues on next page)

¹⁸⁶ <http://www.ada-auth.org/standards/22rm/html/RM-6-5.html>

¹⁸⁷ <https://www.adacore.com/gems/ada-gem-10>

(continued from previous page)

```

25     else
26         Result := S;
27     end if;
28
29     end return;
30 end Make_String;
31
32 S1 : String := "Ada";
33 S2 : String := "Make_With_";
34 begin
35     Put_Line ("No prefix:  "
36             & Make_String (S1, S2, False));
37     Put_Line ("With prefix: "
38             & Make_String (S1, S2, True));
39 end Show_String_Construct;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Statements.Extended_Return_Statements.
↳ Extended_Return_Other_Usages
MD5: a2b26ceed06a0ab66aff6c2b59c02003

Runtime output

```

No prefix:  Ada
With prefix: Make_With_Ada

```

In this example, we first calculate the length of the string and store it in `Length`. We then use this information to initialize the return object of the `Make_String` function.

26.3 Subprograms

26.3.1 Parameter Modes and Associations

In this section, we discuss some details about parameter modes and associations. First of all, as we know, parameters can be either formal or actual:

- Formal parameters are the ones we see in a subprogram declaration and implementation;
- Actual parameters are the ones we see in a subprogram call.
 - Note that actual parameters are also called *subprogram arguments* in other languages.

We define parameter associations as the connection between an actual parameter in a subprogram call and its declaration as a formal parameter in a subprogram specification or body.

In the Ada Reference Manual

- 6.2 Formal Parameter Modes¹⁸⁸
- 6.4.1 Parameter Associations¹⁸⁹

¹⁸⁸ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

¹⁸⁹ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

Formal Parameter Modes

We already discussed formal parameter modes in the *Introduction to Ada* (page 28) course:

in	Parameter can only be read, not written
out	Parameter can be written to, then read
in out	Parameter can be both read and written

As this topic was already discussed in that course — and we used parameter modes extensively in all code examples from that course —, we won't introduce the topic again here. Instead, we'll look into some of the more advanced details.

By-copy and by-reference

In the *Introduction to Ada* (page 28) course, we saw that parameter modes don't correspond directly to how parameters are actually passed. In fact, an **in out** parameter could be passed by copy. For example:

Listing 1051: check_param_passing.ads

```
1 with System;
2
3 procedure Check_Param_Passing
4   (Formal : System.Address;
5    Actual  : System.Address);
```

Listing 1052: check_param_passing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with System.Address_Image;
3
4 procedure Check_Param_Passing
5   (Formal : System.Address;
6    Actual  : System.Address) is
7 begin
8   Put_Line ("Formal parameter at "
9             & System.Address_Image (Formal));
10  Put_Line ("Actual parameter at "
11            & System.Address_Image (Actual));
12  if System.Address_Image (Formal) =
13     System.Address_Image (Actual)
14  then
15    Put_Line
16      ("Parameter is passed by reference.");
17  else
18    Put_Line
19      ("Parameter is passed by copy.");
20  end if;
21 end Check_Param_Passing;
```

Listing 1053: machine_x.ads

```
1 with System;
2
3 package Machine_X is
4
5   procedure Update_Value
6     (V : in out Integer;
7      AV : System.Address);
```

(continues on next page)

(continued from previous page)

```

8
9 end Machine_X;

```

Listing 1054: machine_x.adb

```

1 with Check_Param_Passing;
2
3 package body Machine_X is
4
5     procedure Update_Value
6         (V : in out Integer;
7          AV : System.Address) is
8     begin
9         V := V + 1;
10        Check_Param_Passing (Formal => V'Address,
11                             Actual => AV);
12    end Update_Value;
13
14 end Machine_X;

```

Listing 1055: show_by_copy_by_ref_params.adb

```

1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4     A : Integer := 5;
5 begin
6     Update_Value (A, A'Address);
7 end Show_By_Copy_By_Ref_Params;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↳Associations.By_Copy_By_Ref_Params
MD5: e437d3432703124496f0a217177959eb

```

Runtime output

```

Formal parameter at 00007FFE451EDBCC
Actual parameter at 00007FFE451EDBEC
Parameter is passed by copy.

```

As we can see by running this example,

- the integer variable A in the Show_By_Copy_By_Ref_Params procedure

and

- the V parameter in the Update_Value procedure

have different addresses, so they are different objects. Therefore, we conclude that this parameter is being passed by value, even though it has the **in out** mode. (We talk more about addresses and the `'Address` attribute *later on* (page 397)).

As we know, when a parameter is passed by copy, it is first copied to a temporary object. In the case of a parameter with **in out** mode, the temporary object is copied back to the original (actual) parameter at the end of the subprogram call. In our example, the temporary object indicated by V is copied back to A at the end of the call to Update_Value.

In Ada, it's not the parameter mode that determines whether a parameter is passed by copy or by reference, but rather its type. We can distinguish between three categories:

1. By-copy types;

2. By-reference types;
3. *Unspecified* types.

Obviously, parameters of by-copy types are passed by copy and parameters of by-reference type are passed by reference. However, if a category isn't specified — i.e. when the type is neither a by-copy nor a by-reference type —, the decision is essentially left to the compiler.

As a rule of thumb, we can say that;

- elementary types — and any type that is essentially elementary, such as a private type whose full view is an elementary type — are passed by copy;
- tagged and explicitly limited types — and other types that are essentially tagged, such as task types — are passed by reference.

The following table provides more details:

Type category	Parameter passing	List of types
By copy	By copy	• Elementary types • Descendant of a private type whose full type is a by-copy type
By reference	By reference	• Tagged types • Task and protected types • Explicitly limited record types • Composite types with at least one subcomponent of a by-reference type • Private types whose full type is a by-reference type • Any descendant of the types mentioned above
Unspecified	Either by copy or by reference	• Any type not mentioned above

Note that, for parameters of limited types, only those parameters whose type is *explicitly* limited are always passed by reference. We discuss this topic in more details [in another chapter](#) (page 1116).

Let's see an example:

Listing 1056: machine_x.ads

```
1 with System;
2
3 package Machine_X is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
```

(continues on next page)

(continued from previous page)

```

7
8  type Rec is record
9      A : Integer;
10 end record;
11
12 type Rec_Array is record
13     A : Integer;
14     Arr : Integer_Array (1 .. 100);
15 end record;
16
17 type Tagged_Rec is tagged record
18     A : Integer;
19 end record;
20
21 procedure Update_Value
22     (R : in out Rec;
23      AR : System.Address);
24
25 procedure Update_Value
26     (RA : in out Rec_Array;
27      ARA : System.Address);
28
29 procedure Update_Value
30     (R : in out Tagged_Rec;
31      AR : System.Address);
32
33 end Machine_X;

```

Listing 1057: machine_x.adb

```

1  with Check_Param_Passing;
2
3  package body Machine_X is
4
5      procedure Update_Value
6          (R : in out Rec;
7           AR : System.Address)
8      is
9      begin
10         R.A := R.A + 1;
11         Check_Param_Passing (Formal => R'Address,
12                             Actual => AR);
13     end Update_Value;
14
15     procedure Update_Value
16         (RA : in out Rec_Array;
17          ARA : System.Address)
18     is
19     begin
20         RA.A := RA.A + 1;
21         Check_Param_Passing (Formal => RA'Address,
22                             Actual => ARA);
23     end Update_Value;
24
25     procedure Update_Value
26         (R : in out Tagged_Rec;
27          AR : System.Address)
28     is
29     begin
30         R.A := R.A + 1;
31         Check_Param_Passing (Formal => R'Address,

```

(continues on next page)

(continued from previous page)

```

32             Actual => AR);
33     end Update_Value;
34
35 end Machine_X;

```

Listing 1058: show_by_copy_by_ref_params.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Machine_X;   use Machine_X;
3
4  procedure Show_By_Copy_By_Ref_Params is
5      TR : Tagged_Rec := (A => 5);
6      R  : Rec       := (A => 5);
7      RA : Rec_Array := (A => 5,
8                          Arr => (others => 0));
9  begin
10     Put_Line ("Tagged record");
11     Update_Value (TR, TR'Address);
12
13     Put_Line ("Untagged record");
14     Update_Value (R, R'Address);
15
16     Put_Line ("Untagged record with array");
17     Update_Value (RA, RA'Address);
18 end Show_By_Copy_By_Ref_Params;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
 ↳Associations.By_Copy_By_Ref_Params
 MD5: 3ca46380c4df36af9393041181ff2f17

Runtime output

```

Tagged record
Formal parameter at 00007FFC509E3270
Actual parameter at 00007FFC509E3270
Parameter is passed by reference.
Untagged record
Formal parameter at 00007FFC509E30BC
Actual parameter at 00007FFC509E326C
Parameter is passed by copy.
Untagged record with array
Formal parameter at 00007FFC509E30D0
Actual parameter at 00007FFC509E30D0
Parameter is passed by reference.

```

When we run this example, we see that the object of tagged type (Tagged_Rec) is passed by reference to the Update_Value procedure. In the case of the objects of untagged record types, you might see this:

- the parameter of Rec type — which is an untagged record with a single component of integer type —, the parameter is passed by copy;
- the parameter of Rec_Array type — which is an untagged record with a large array of 100 components —, the parameter is passed by reference.

Because Rec and Rec_Array are neither by-copy nor by-reference types, the decision about how to pass them to the Update_Value procedure is made by the compiler. (Thus, it is possible that you see different results when running the code above.)

Bounded errors

When we use parameters of types that are neither by-copy nor by-reference types, we might encounter the situation where we have the same object bound to different names in a subprogram. For example, if:

- we use a global object `Global_R` of a record type `Rec`

and

- we have a subprogram with an in-out parameter of the same record type `Rec`

and

- we pass `Global_R` as the actual parameter for the in-out parameter of this subprogram,

then we have two access paths to this object: one of them using the global variable directly, and the other one using it indirectly via the in-out parameter. This situation could lead to undefined behavior or to a program error. Consider the following code example:

Listing 1059: machine_x.ads

```

1 with System;
2
3 package Machine_X is
4
5     type Rec is record
6         A : Integer;
7     end record;
8
9     Global_R : Rec := (A => 0);
10
11     procedure Update_Value
12         (R : in out Rec;
13          AR : System.Address);
14
15 end Machine_X;
```

Listing 1060: machine_x.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2
3 with Check_Param_Passing;
4
5 package body Machine_X is
6
7     procedure Update_Value
8         (R : in out Rec;
9          AR : System.Address)
10     is
11         procedure Show_Vars is
12             begin
13                 Put_Line ("Global_R.A: "
14                           & Integer'Image (Global_R.A));
15                 Put_Line ("R.A: "
16                           & Integer'Image (R.A));
17             end Show_Vars;
18         begin
19             Check_Param_Passing (Formal => R'Address,
20                                 Actual => AR);
21
22             Put_Line ("Incrementing Global_R.A...");
23             Global_R.A := Global_R.A + 1;
```

(continues on next page)

(continued from previous page)

```

24     Show_Vars;
25
26     Put_Line ("Incrementing R.A...");
27     R.A := R.A + 5;
28     Show_Vars;
29     end Update_Value;
30
31 end Machine_X;

```

Listing 1061: show_by_copy_by_ref_params.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Machine_X;   use Machine_X;
3
4  procedure Show_By_Copy_By_Ref_Params is
5  begin
6      Put_Line ("Calling Update_Value...");
7      Update_Value (Global_R, Global_R'Address);
8
9      Put_Line ("After call to Update_Value...");
10     Put_Line ("Global_R.A: "
11         & Integer'Image (Global_R.A));
12 end Show_By_Copy_By_Ref_Params;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↳ Associations.By_Copy_By_Ref_Params
MD5: 96be7054b7ff64a304705edf6b15f031

Runtime output

```

Calling Update_Value...
Formal parameter at 00007FFFEAAFF32C
Actual parameter at 0000581CD3E1F3C4
Parameter is passed by copy.
Incrementing Global_R.A...
Global_R.A: 1
R.A:      0
Incrementing R.A...
Global_R.A: 1
R.A:      5
After call to Update_Value...
Global_R.A: 5

```

In the `Update_Value` procedure, because `Global_R` and `R` have a type that is neither a by-pass nor a by-reference type, the language does not specify whether the old or the new value would be read in the calls to `Put_Line`. In other words, the actual behavior is undefined. Also, this situation might raise the `Program_Error` exception.

Important

As a general advice:

- you should be very careful when using global variables and
- you should avoid passing them as parameters in situations such as the one illustrated in the code example above.

Aliased parameters

When a parameter is specified as *aliased*, it is always passed by reference, independently of the type we're using. In this sense, we can use this keyword to circumvent the rules mentioned so far. (We discuss more about *aliasing* (page 922) and *aliased parameters* (page 930) later on.)

Let's rewrite a previous code example that has a parameter of elementary type and change it to *aliased*:

Listing 1062: machine_x.ads

```

1 with System;
2
3 package Machine_X is
4
5     procedure Update_Value
6         (V : aliased in out Integer;
7          AV : System.Address);
8
9 end Machine_X;
```

Listing 1063: machine_x.adb

```

1 with Check_Param_Passing;
2
3 package body Machine_X is
4
5     procedure Update_Value
6         (V : aliased in out Integer;
7          AV : System.Address)
8     is
9     begin
10         V := V + 1;
11         Check_Param_Passing (Formal => V'Address,
12                             Actual => AV);
13     end Update_Value;
14
15 end Machine_X;
```

Listing 1064: show_by_copy_by_ref_params.adb

```

1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4     A : aliased Integer := 5;
5 begin
6     Update_Value (A, A'Address);
7 end Show_By_Copy_By_Ref_Params;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
 ↳Associations.By_Copy_By_Ref_Params
 MD5: c066af3a7081815d0a7598733f9e6aec

Runtime output

Formal parameter at 00007FFE676106AC
 Actual parameter at 00007FFE676106AC
 Parameter is passed by reference.

As we can see, A is now passed by reference.

Note that we can only pass aliased objects to aliased parameters. If we try to pass a non-aliased object, we get a compilation error:

Listing 1065: show_by_copy_by_ref_params.adb

```
1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4   A : Integer := 5;
5 begin
6   Update_Value (A, A'Address);
7 end Show_By_Copy_By_Ref_Params;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
↳Associations.By_Copy_By_Ref_Params
MD5: 9e6586e0b771de68040131cae81799b8

Build output

```
show_by_copy_by_ref_params.adb:6:18: error: actual for aliased formal "V" must be_
↳aliased object
gprbuild: *** compilation phase failed
```

Again, we discuss more about *aliased parameters* (page 930) and *aliased objects* (page 924) later on in the context of access types.

Parameter Associations

When actual parameters are associated with formal parameters, some rules are checked. As a typical example, the type of each actual parameter must match the type of the corresponding actual parameter. In this section, we see some details about how this association is made and some of the potential errors.

In the Ada Reference Manual

- 6.4.1 Parameter Associations¹⁹⁰

Parameter order and association

As we already know, when calling subprograms, we can use positional or named parameter association — or a mixture of both. Also, parameters can have default values. Let's see some examples:

Listing 1066: operations.ads

```
1 package Operations is
2
3   procedure Add (Left  : in out Integer;
4                 Right :          Float := 1.0);
5
6 end Operations;
```

¹⁹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

Listing 1067: operations.adb

```

1 package body Operations is
2
3   procedure Add (Left  : in out Integer;
4                 Right :          Float := 1.0) is
5   begin
6     Left := Left + Integer (Right);
7   end Add;
8
9 end Operations;

```

Listing 1068: show_param_association.adb

```

1 with Operations; use Operations;
2
3 procedure Show_Param_Association is
4   A : Integer := 5;
5 begin
6   -- Positional association
7   Add (A, 2.0);
8
9   -- Positional association
10  -- (using default value)
11  Add (A);
12
13  -- Named association
14  Add (Left => A,
15       Right => 2.0);
16
17  -- Named association (inversed order)
18  Add (Right => 2.0,
19       Left  => A);
20
21  -- Mixed positional / named association
22  Add (A, Right => 2.0);
23 end Show_Param_Association;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
 ↳Associations.Param_Association_1
 MD5: 64d3f44ac2bf72317fae22658f6d218e

This code snippet has examples of positional and name parameter association. Also, it has an example of mixed positional / named parameter association. In most cases, the actual A parameter is associated with the formal Left parameter, and the actual 2.0 parameter is associated with the formal Right parameter.

In addition to that, parameters can have default values, so, when we write `Add (A)`, the A variable is associated with the Left parameter and the default value (1.0) is associated with the Right parameter.

Also, when we use named parameter association, the parameter order is irrelevant: we can, for example, write the last parameter as the first one. Therefore, we can write `Add (Right => 2.0, Left => A)` instead of `Add (Left => A, Right => 2.0)`.

Ambiguous calls

Ambiguous calls can be detected by the compiler during parameter association. For example, when we have both default values in parameters and subprogram overloading, the compiler might be unable to decide which subprogram we're calling:

Listing 1069: operations.ads

```
1 package Operations is
2
3   procedure Add (Left : in out Integer);
4
5   procedure Add (Left : in out Integer;
6                 Right : Float := 1.0);
7
8 end Operations;
```

Listing 1070: operations.adb

```
1 package body Operations is
2
3   procedure Add (Left : in out Integer) is
4   begin
5     Left := Left + 1;
6   end Add;
7
8   procedure Add (Left : in out Integer;
9                 Right : Float := 1.0) is
10  begin
11    Left := Left + Integer (Right);
12  end Add;
13
14 end Operations;
```

Listing 1071: show_param_association.adb

```
1 with Operations; use Operations;
2
3 procedure Show_Param_Association is
4   A : Integer := 5;
5 begin
6   Add (A);
7   -- ERROR: cannot decide which
8   --       procedure to take
9 end Show_Param_Association;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
Associations.Param_Association_1
MD5: 2725517f82d4068b669028eca1815079

Build output

```
show_param_association.adb:6:04: error: ambiguous expression (cannot resolve "Add")
show_param_association.adb:6:04: error: possible interpretation at operations.ads:5
show_param_association.adb:6:04: error: possible interpretation at operations.ads:3
gprbuild: *** compilation phase failed
```

As we see in this example, the Add procedure is overloaded. The first instance has one parameter, and the second instance has two parameters, where the second parameter has a default value. When we call Add with just one parameter, the compiler cannot decide

whether we intend to call

- the first instance of Add with one parameter

or

- the second instance of Add using the default value for the second parameter.

In this specific case, there are multiple options to solve the issue, but all of them involve redesigning the package specification:

- we could just rename one of Add procedures (thereby eliminating the subprogram overloading);
- we could rename the first parameter of one of the Add procedures and use named parameter association in the call to the procedure;
 - For example, we could rename the parameter to Value and call Add (Value => A).
- remove the default value from the second parameter of the second instance of Add.

Overlapping actual parameters

When we have more than one **out** or **in out** parameters in a subprogram, we might run into the situation where the actual parameter overlaps with another parameter. For example:

Listing 1072: machine_x.ads

```

1 package Machine_X is
2
3     procedure Update_Value (V1 : in out Integer;
4                             V2 : out Integer);
5
6 end Machine_X;
```

Listing 1073: machine_x.adb

```

1 package body Machine_X is
2
3     procedure Update_Value (V1 : in out Integer;
4                             V2 : out Integer) is
5     begin
6         V1 := V1 + 1;
7         V2 := V2 + 1;
8     end Update_Value;
9
10 end Machine_X;
```

Listing 1074: show_by_copy_by_ref_params.adb

```

1 with Machine_X; use Machine_X;
2
3 procedure Show_By_Copy_By_Ref_Params is
4     A : Integer := 5;
5 begin
6     Update_Value (A, A);
7 end Show_By_Copy_By_Ref_Params;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Parameter_Modes_
 Associations.Illegal_Calls
 MD5: d18a7056463fee9298dd1fdef0a31daf

Build output

```
show_by_copy_by_ref_params.adb:6:18: error: writable actual for "V1" overlaps with
↳ actual for "V2"
gprbuild: *** compilation phase failed
```

In this case, we're using A for both output parameters in the call to `Update_Value`. Passing one variable to more than one output parameter in a given call is forbidden in Ada, so this triggers a compilation error. Depending on the specific context, you could solve this issue by using temporary variables for the other output parameters.

26.3.2 Operators

Operators are commonly used for variables of scalar types such as **Integer** and **Float**. In these cases, they replace *usual* function calls. (To be more precise, operators are function calls, but written in a different format.) For example, we simply write `A := A + B + C`; when we want to add three integer variables. A hypothetical, non-intuitive version of this operation could be `A := Add (Add (A, B), C)`; . In such cases, operators allow for expressing function calls in a more intuitive way.

Many primitive operators exist for scalar types. We classify them as follows:

Category	Operators
Logical	and, or, xor
Relational	<code>=, /=, <, <=, >, >=</code>
Unary adding	<code>+, -</code>
Binary adding	<code>+, -, &</code>
Multiplying	<code>*, /, mod, rem</code>
Highest precedence	<code>**</code> , abs, not

** In the Ada Reference Manual**

- [4.5 Operators and Expression Evaluation](#)¹⁹¹

User-defined operators

For non-scalar types, not all operators are defined. For example, it wouldn't make sense to expect a compiler to include an addition operator for a record type with multiple components. Exceptions to this rule are the equality and inequality operators (`=` and `/=`), which are defined for any type (be it scalar, record types, and array types).

For array types, the concatenation operator (`&`) is a primitive operator:

Listing 1075: `integer_arrays.ads`

```
1 package Integer_Arrays is
2
3     type Integer_Array is
4       array (Positive range <>) of Integer;
5
6 end Integer_Arrays;
```

¹⁹¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-5.html>

Listing 1076: show_array_concatenation.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Integer_Arrays; use Integer_Arrays;
3
4  procedure Show_Array_Concatenation is
5      A, B : Integer_Array (1 .. 5);
6      R    : Integer_Array (1 .. 10);
7  begin
8      A := (1 & 2 & 3 & 4 & 5);
9      B := (6 & 7 & 8 & 9 & 10);
10     R := A & B;
11
12     for E of R loop
13         Put (E'Image & ' ');
14     end loop;
15     New_Line;
16 end Show_Array_Concatenation;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_Concat
MD5: 1899e66ec1d0b36b10d8b89fc2dfac0e

Runtime output

```
1 2 3 4 5 6 7 8 9 10
```

In this example, we're using the primitive & operator to concatenate the A and B arrays in the assignment to R. Similarly, we're concatenating individual components (integer values) to create an aggregate that we assign to A and B.

In contrast to this, the addition operator is not available for arrays:

Listing 1077: integer_arrays.ads

```

1  package Integer_Arrays is
2
3      type Integer_Array is
4          array (Positive range <>) of Integer;
5
6  end Integer_Arrays;

```

Listing 1078: show_array_addition.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Integer_Arrays; use Integer_Arrays;
3
4  procedure Show_Array_Addition is
5      A, B, R : Integer_Array (1 .. 5);
6  begin
7      A := (1 & 2 & 3 & 4 & 5);
8      B := (6 & 7 & 8 & 9 & 10);
9      R := A + B;
10
11     for E of R loop
12         Put (E'Image & ' ');
13     end loop;
14     New_Line;
15
16 end Show_Array_Addition;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_
↳Addition
MD5: d94f9791523359d390a7cafd900d1268
```

Build output

```
show_array_addition.adb:9:11: error: there is no applicable operator "+" for type
↳"Integer_Array" defined at integer_arrays.ads:3
gprbuild: *** compilation phase failed
```

We can, however, define *custom* operators for any type. For example, if a specific type doesn't have a predefined addition operator, we can define our own + operator for it.

Note that we're limited to the operator symbols that are already defined by the Ada language (see the previous table for the complete list of operators). In other words, the operator we define must be selected from one of those existing symbols; we cannot use new symbols for custom operators.

In other languages

Some programming languages — such as Haskell — allow you to define and use custom operator symbols. For example, in Haskell, you can create a new "broken bar" (!) operator for integer values:

```
(!) :: Int -> Int -> Int
a ! b = a + a + b

main = putStrLn $ show (2 ! 3)
```

This is not possible in Ada.

Let's define a custom addition operator that adds individual components of the `Integer_Array` type:

Listing 1079: integer_arrays.ads

```
1 package Integer_Arrays is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     function "+" (Left, Right : Integer_Array)
7         return Integer_Array
8     with Post =>
9         (for all I in "+"'Result'Range =>
10          "+"'Result (I) = Left (I) + Right (I));
11
12 end Integer_Arrays;
```

Listing 1080: integer_arrays.adb

```
1 package body Integer_Arrays is
2
3     function "+" (Left, Right : Integer_Array)
4         return Integer_Array
5     is
6         R : Integer_Array (Left'Range);
7     begin
8         for I in Left'Range loop
```

(continues on next page)

(continued from previous page)

```

9      R (I) := Left (I) + Right (I);
10     end loop;
11
12     return R;
13 end "+";
14
15 end Integer_Arrays;

```

Listing 1081: show_array_addition.adb

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Integer_Arrays; use Integer_Arrays;
3
4 procedure Show_Array_Addition is
5   A, B, R : Integer_Array (1 .. 5);
6 begin
7   A := (1 & 2 & 3 & 4 & 5);
8   B := (6 & 7 & 8 & 9 & 10);
9   R := A + B;
10
11   for E of R loop
12     Put (E'Image & ' ');
13   end loop;
14   New_Line;
15
16 end Show_Array_Addition;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Integer_Arrays_Addition
 MD5: 6f50fa47270d97d3fb50379b6275777d

Runtime output

```
7 9 11 13 15
```

Now, the `R := A + B` line doesn't trigger a compilation error anymore because the `+` operator is defined for the `Integer_Array` type.

In the implementation of the `+`, we return an array with the range of the `Left` array where each component is the sum of the `Left` and `Right` arrays. In the declaration of the `+` operator, we're defining the expected behavior in the postcondition. Here, we're saying that, for each index of the resulting array (**for all** `I` **in** `"+"Result'Range`), the value of each component of the resulting array at that specific index is the sum of the components from the `Left` and `Right` arrays at the same index (`"+"Result (I) = Left (I) + Right (I)`). (**for all** denotes a *quantified expression* (page 734).)

Note that, in this implementation, we assume that the range of `Right` is a subset of the range of `Left`. If that is not the case, the `Constraint_Error` exception will be raised at runtime in the loop. (You can test this by declaring `B` as `Integer_Array (5 .. 10)`, for example.)

We can also define custom operators for record types. For example, we could declare two `+` operators for a record containing the name and address of a person:

Listing 1082: addresses.ads

```

1 package Addresses is
2
3   type Person is private;

```

(continues on next page)

(continued from previous page)

```

4
5  function "+" (Name    : String;
6               Address : String)
7               return Person;
8  function "+" (Left, Right : Person)
9               return Person;
10
11  procedure Display (P : Person);
12
13  private
14
15  subtype Name_String  is String (1 .. 40);
16  subtype Address_String is String (1 .. 100);
17
18  type Person is record
19      Name    : Name_String;
20      Address : Address_String;
21  end record;
22
23  end Addresses;

```

Listing 1083: addresses.adb

```

1  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  package body Addresses is
5
6      function "+" (Name    : String;
7                   Address : String)
8                   return Person
9
10     is
11     begin
12         return (Name =>
13                 Head (Name,
14                       Name_String'Length),
15                 Address =>
16                 Head (Address,
17                       Address_String'Length));
18     end "+";
19
20     function "+" (Left, Right : Person)
21                 return Person
22
23     is
24     begin
25         return (Name => Left.Name,
26                 Address => Right.Address);
27     end "+";
28
29     procedure Display (P : Person) is
30     begin
31         Put_Line ("Name: " & P.Name);
32         Put_Line ("Address: " & P.Address);
33         New_Line;
34     end Display;
35
36  end Addresses;

```

Listing 1084: show_address_addition.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Addresses;   use Addresses;
3
4 procedure Show_Address_Addition is
5     John : Person := "John" + "4 Main Street";
6     Jane : Person := "Jane" + "7 High Street";
7 begin
8     Display (John);
9     Display (Jane);
10    Put_Line ("-----");
11
12    Jane := Jane + John;
13    Display (Jane);
14 end Show_Address_Addition;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Rec_Operator
MD5: c69ff43ed5a80a0c62bad87eada14301

Runtime output

```

Name:    John
Address: 4 Main Street

Name:    Jane
Address: 7 High Street

-----
Name:    Jane
Address: 4 Main Street

```

In this example, the first `+` operator takes two strings — with the name and address of a person — and returns an object of `Person` type. We use this operator to initialize the `John` and `Jane` variables.

The second `+` operator in this example brings two people together. Here, the person on the left side of the `+` operator moves to the home of the person on the right side. In this specific case, Jane is moving to John's house.

As a small remark, we usually expect that the `+` operator is commutative. In other words, changing the order of the elements in the operation doesn't change the result. However, in our definition above, this is *not* the case, as we can confirm by comparing the operation in both orders:

Listing 1085: show_address_addition.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Addresses;   use Addresses;
3
4 procedure Show_Address_Addition is
5     John : constant Person :=
6         "John" + "4 Main Street";
7     Jane : constant Person :=
8         "Jane" + "7 High Street";
9 begin
10    if Jane + John = John + Jane then
11        Put_Line ("It's commutative!");

```

(continues on next page)

(continued from previous page)

```

12     else
13         Put_Line ("It's not commutative!");
14     end if;
15 end Show_Address_Addition;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operators.Rec_Operator
MD5: 2af6e1a31100a1d0fa786d42cc93c09b

Runtime output

It's not commutative!

In this example, we're using the primitive = operator for the Person to assess whether the result of the addition is commutative.

In the Ada Reference Manual

- 6.1 Subprogram Declarations¹⁹²

26.3.3 Expression functions

Usually, we implement Ada functions with a construct like this: **begin return X; end;**. In other words, we create a **begin ... end;** block and we have at least one **return** statement in that block. An expression function, in contrast, is a function that is implemented with a simple expression in parentheses, such as (X);. In this case, we don't use a **begin ... end;** block or a **return** statement.

As an example of an expression, let's say we want to implement a function named `Is_Zero` that checks if the value of the integer parameter `I` is zero. We can implement this function with the expression `I = 0`. In the usual approach, we would create the implementation by writing **is begin return I = 0; end Is_Zero;**. When using expression functions, however, we can simplify the implementation by just writing **is (I = 0);**. This is the complete code of `Is_Zero` using an expression function:

Listing 1086: `expr_func.ads`

```

1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4                     return Boolean is
5         (I = 0);
6
7 end Expr_Func;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_Expression_Function_1
MD5: 44779999566f764279e1c2f292226f95

An expression function has the same effect as the usual version using a block. In fact, the code above is similar to this implementation of the `Is_Zero` function using a block:

¹⁹² <http://www.ada-auth.org/standards/22rm/html/RM-6-1.html>

Listing 1087: expr_func.ads

```

1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean;
5
6 end Expr_Func;
```

Listing 1088: expr_func.adb

```

1 package body Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5     begin
6         return I = 0;
7     end Is_Zero;
8
9 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_Expression_Function_2
 MD5: 4d90b1c63928cbaf9c86a6cc6421bb61

The only difference between these two versions of the Expr_Func packages is that, in the first version, the package specification contains the implementation of the Is_Zero function, while, in the second version, the implementation is in the body of the Expr_Func package.

An expression function can be, at same time, the specification and the implementation of a function. Therefore, in the first version of the Expr_Func package above, we don't have a separate implementation of the Is_Zero function because (I = 0) is the actual implementation of the function. Note that this is only possible for expression functions; you cannot have a function implemented with a block in a package specification. For example, the following code is wrong and won't compile:

Listing 1089: expr_func.ads

```

1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5     begin
6         return I = 0;
7     end Is_Zero;
8
9 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_Expression_Function_3
 MD5: 919f9c101b3224006e1302130eba8dd2

We can, of course, separate the function declaration from its implementation as an expression function. For example, we can rewrite the first version of the Expr_Func package and move the expression function to the body of the package:

Listing 1090: expr_func.ads

```
1 package Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean;
5
6 end Expr_Func;
```

Listing 1091: expr_func.adb

```
1 package body Expr_Func is
2
3     function Is_Zero (I : Integer)
4         return Boolean is
5         (I = 0);
6
7 end Expr_Func;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Simple_Expression_Function_4
↪ Expression_Function_4
MD5: 491a491da92636a35579f870969aaf08

In addition, we can use expression functions in the private part of a package specification. For example, the following code declares the `Is_Valid` function in the specification of the `My_Data` package, while its implementation is an expression function in the private part of the package specification:

Listing 1092: my_data.ads

```
1 package My_Data is
2
3     type Data is private;
4
5     function Is_Valid (D : Data)
6         return Boolean;
7
8 private
9
10    type Data is record
11        Valid : Boolean;
12    end record;
13
14    function Is_Valid (D : Data)
15        return Boolean is
16        (D.Valid);
17
18 end My_Data;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.Private_Expression_Function_1
↪ Private_Expression_Function_1
MD5: beb57eca67b3954097e0f7ac00ea70c9

Naturally, we could write the function implementation in the package body instead:

Listing 1093: my_data.ads

```

1 package My_Data is
2
3     type Data is private;
4
5     function Is_Valid (D : Data)
6         return Boolean;
7
8 private
9
10    type Data is record
11        Valid : Boolean;
12    end record;
13
14 end My_Data;
```

Listing 1094: my_data.adb

```

1 package body My_Data is
2
3     function Is_Valid (D : Data)
4         return Boolean is
5         (D.Valid);
6
7 end My_Data;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Expression_Functions.
 ↳ Private_Expression_Function_2
 MD5: 3c6e2a3c53c7c8e1a7b86efccdc3bf8d

** In the Ada Reference Manual**

- 6.8 Expression functions¹⁹³

26.3.4 Overloading

** Note**

This section was originally written by Robert A. Duff and published as [Gem #50: Overload Resolution](#)¹⁹⁴.

Ada allows overloading of subprograms, which means that two or more subprogram declarations with the same name can be visible at the same place. Here, "name" can refer to operator symbols, like "+". Ada also allows overloading of various other notations, such as literals and aggregates.

In most languages that support overloading, overload resolution is done "bottom up" — that is, information flows from inner constructs to outer constructs. As usual, computer folks draw their trees upside-down, with the root at the top. For example, if we have two procedures Print:

¹⁹³ <http://www.ada-auth.org/standards/22rm/html/RM-6-8.html>

¹⁹⁴ <https://www.adacore.com/gems/gem-50>

Listing 1095: show_overloading.adb

```

1  procedure Show_Overloading is
2
3      package Types is
4          type Sequence is null record;
5          type Set is null record;
6
7          procedure Print (S : Sequence) is null;
8          procedure Print (S : Set) is null;
9      end Types;
10
11     use Types;
12
13     X : Sequence;
14 begin
15
16     -- Compiler selects Print (S : Sequence)
17     Print (X);
18 end Show_Overloading;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
MD5: 020c4f04285c80c1050d8edbaef2dbcae

the type of X determines which Print is meant in the call.

Ada is unusual in that it supports top-down overload resolution as well:

Listing 1096: show_top_down_overloading.adb

```

1  procedure Show_Top_Down_Overloading is
2
3      package Types is
4          type Sequence is null record;
5          type Set is null record;
6
7          function Empty return Sequence is
8              ((others => <>));
9
10         function Empty return Set is
11             ((others => <>));
12
13         procedure Print_Sequence (S : Sequence) is
14             null;
15
16         procedure Print_Set (S : Set) is
17             null;
18     end Types;
19
20     use Types;
21
22     X : Sequence;
23 begin
24     -- Compiler selects function
25     -- Empty return Sequence
26     Print_Sequence (Empty);
27 end Show_Top_Down_Overloading;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
 MD5: 3b776a3efdee3d7e583ddb5f5159c9a1b

The type of the formal parameter `S` of `Print_Sequence` determines which `Empty` is meant in the call. In C++, for example, the equivalent of the `Print (X)` example would resolve, but the `Print_Sequence (Empty)` would be illegal, because C++ does not use top-down information.

If we overload things too heavily, we can cause ambiguities:

Listing 1097: show_overloading_error.adb

```

1 procedure Show_Overloading_Error is
2
3   package Types is
4     type Sequence is null record;
5     type Set is null record;
6
7     function Empty return Sequence is
8       ((others => <>));
9
10    function Empty return Set is
11      ((others => <>));
12
13    procedure Print (S : Sequence) is
14      null;
15
16    procedure Print (S : Set) is
17      null;
18  end Types;
19
20  use Types;
21
22  X : Sequence;
23 begin
24   Print (Empty); -- Illegal!
25 end Show_Overloading_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Overloading
 MD5: 5182c517a1afff4568ab2404ac66fda8

Build output

```

show_overloading_error.adb:24:04: error: ambiguous expression (cannot resolve
↳ "Print")
show_overloading_error.adb:24:04: error: possible interpretation at line 16
show_overloading_error.adb:24:04: error: possible interpretation at line 13
show_overloading_error.adb:24:11: error: ambiguous call to "Empty"
show_overloading_error.adb:24:11: error: possible interpretation at line 10
show_overloading_error.adb:24:11: error: possible interpretation at line 7
gprbuild: *** compilation phase failed
```

The call is ambiguous, and therefore illegal, because there are two possible meanings. One way to resolve the ambiguity is to use a qualified expression to say which type we mean:

```
Print (Sequence'(Empty));
```

Note that we're now using both bottom-up and top-down overload resolution: `Sequence'` determines which `Empty` is meant (top down) and which `Print` is meant (bottom up). You can qualify an expression, even if it is not ambiguous according to Ada rules — you might

want to clarify the type because it might be ambiguous for human readers.

Of course, you could instead resolve the `Print (Empty)` example by modifying the source code so the names are unique, as in the earlier examples. That might well be the best solution, assuming you can modify the relevant sources. Too much overloading can be confusing. How much is "too much" is in part a matter of taste.

Ada really needs to have top-down overload resolution, in order to resolve literals. In some languages, you can tell the type of a literal by looking at it, for example appending `L` (letter `el`) means "the type of this literal is long int". That sort of kludge won't work in Ada, because we have an open-ended set of integer types:

Listing 1098: `show_literal_resolution.adb`

```
1 procedure Show_Literal_Resolution is
2
3     type Apple_Count is range 0 .. 100;
4
5     procedure Peel (Count : Apple_Count) is null;
6 begin
7     Peel (20);
8 end Show_Literal_Resolution;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Literal_
↳Resolution
MD5: f428b6b4c642c44ede6bc21e7522c532

You can't tell by looking at the literal `20` what its type is. The type of formal parameter **Count** tells us that `20` is an `Apple_Count`, as opposed to some other type, such as `Standard.Long_Integer`.

Technically, the type of `20` is `universal_integer`, which is implicitly converted to `Apple_Count` — it's really the result type of that implicit conversion that is at issue. But that's an obscure point — you won't go *too* far wrong if you think of the integer literal notation as being overloaded on all integer types.

Developers sometimes wonder why the compiler can't resolve something that seems obvious. For example:

Listing 1099: `show_literal_resolution_error.adb`

```
1 procedure Show_Literal_Resolution_Error is
2
3     type Apple_Count is range 0 .. 100;
4     procedure Slice (Count : Apple_Count) is null;
5
6     type Orange_Count is range 0 .. 10_000;
7     procedure Slice (Count : Orange_Count) is null;
8 begin
9     Slice (Count => (10_000)); -- Illegal!
10 end Show_Literal_Resolution_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Literal_
↳Resolution_Error
MD5: 4789d8eea9b82649ba8e453bb861688a

Build output

```
show_literal_resolution_error.adb:9:04: error: ambiguous expression (cannot
↳ resolve "Slice")
show_literal_resolution_error.adb:9:04: error: possible interpretation at line 7
show_literal_resolution_error.adb:9:04: error: possible interpretation at line 4
gprbuild: *** compilation phase failed
```

This call is ambiguous, and therefore illegal. But why? Clearly the developer must have meant the `Orange_Count` one, because `10_000` is out of range for `Apple_Count`. And all the relevant expressions happen to be static.

Well, a good rule of thumb in language design (for languages with overloading) is that the overload resolution rules should not be "too smart". We want this example to be illegal to avoid confusion on the part of developers reading the code. As usual, a qualified expression fixes it:

```
Slice (Count => Orange_Count'(10_000));
```

Another example, similar to the literal, is the aggregate. Ada uses a simple rule: the type of an aggregate is determined top down (i.e., from the context in which the aggregate appears). Bottom-up information is not used; that is, the compiler does not look inside the aggregate in order to determine its type.

Listing 1100: show_record_resolution_error.adb

```
1 procedure Show_Record_Resolution_Error is
2
3     type Complex is record
4         Re, Im : Float;
5     end record;
6
7     procedure Grind (X : Complex) is null;
8     procedure Grind (X : String) is null;
9 begin
10    Grind (X => (Re => 1.0, Im => 1.0));
11    -- ~~~~~
12    -- Illegal!
13 end Show_Record_Resolution_Error;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Overloading.Record_
↳ Resolution_Error
MD5: e3dd1fd0c403bcf672f4bab881b8ef9
```

Build output

```
show_record_resolution_error.adb:10:04: error: ambiguous expression (cannot
↳ resolve "Grind")
show_record_resolution_error.adb:10:04: error: possible interpretation at line 8
show_record_resolution_error.adb:10:04: error: possible interpretation at line 7
gprbuild: *** compilation phase failed
```

There are two `Grind` procedures visible, so the type of the aggregate could be `Complex` or `String`, so it is ambiguous and therefore illegal. The compiler is not required to notice that there is only one type with components `Re` and `Im`, of some real type — in fact, the compiler is not *allowed* to notice that, for overloading purposes.

We can qualify as usual:

```
Grind (X => Complex'(Re => 1.0, Im => 1.0));
```

Only after resolving that the type of the aggregate is `Complex` can the compiler look inside and make sure `Re` and `Im` make sense.

This not-too-smart rule for aggregates helps prevent confusion on the part of developers reading the code. It also simplifies the compiler, and makes the overload resolution algorithm reasonably efficient.

26.3.5 Operator Overloading

We've seen *previously* (page 774) that we can define custom operators for any type. We've also seen that subprograms can be *overloaded* (page 783). Since operators are functions, we're essentially talking about operator overloading, as we're defining the same operator (say `+` or `-`) for different types.

As another example of operator overloading, in the Ada standard library, operators are defined for the `Complex` type of the `Ada.Numerics.Generic_Complex_Types` package. This package contains not only the definition of the `+` operator for two objects of `Complex` type, but also for combination of `Complex` and other types. For instance, we can find these declarations:

```
function "+" (Left, Right : Complex)
    return Complex;
function "+" (Left : Complex; Right : Real'Base)
    return Complex;
function "+" (Left : Real'Base; Right : Complex)
    return Complex;
```

This example shows that the `+` operator — as well as other operators — are being overloaded in the `Generic_Complex_Types` package.

In the Ada Reference Manual

- 6.6 Overloading of Operators¹⁹⁵
- G.1.1 Complex Types¹⁹⁶

26.3.6 Operator Overriding

We can also override operators of derived types. This allows for modifying the behavior of operators for the corresponding derived types.

To override an operator of a derived type, we simply implement a function for that operator. This is the same as how we implement custom operators (as we've seen previously).

As an example, when adding two fixed-point values, the result might be out of range, which causes an exception to be raised. A common strategy to avoid exceptions in this case is to saturate the resulting value. This strategy is typically employed in signal processing algorithms, for example.

In this example, we declare and use the 32-bit fixed-point type `TQ31`:

Listing 1101: `fixed_point.ads`

```
1 package Fixed_Point is
2
3   D : constant := 2.0 ** (-31);
4   type TQ31 is delta D range -1.0 .. 1.0 - D;
```

(continues on next page)

¹⁹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-6-6.html>

¹⁹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-G-1-1.html>

(continued from previous page)

```

5
6 end Fixed_Point;

```

Listing 1102: show_sat_op.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Fixed_Point; use Fixed_Point;
3
4 procedure Show_Sat_Op is
5     A, B, C : TQ31;
6 begin
7     A := TQ31'Last;
8     B := TQ31'Last;
9     C := A + B;
10
11     Put_Line (A'Image & " + "
12              & B'Image & " = "
13              & C'Image);
14
15     A := TQ31'First;
16     B := TQ31'First;
17     C := A + B;
18
19     Put_Line (A'Image & " + "
20              & B'Image & " = "
21              & C'Image);
22
23 end Show_Sat_Op;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operator_Overriding.Fixed_
 ↪Point_Exception
 MD5: 15d8860773ec7c0e505d0ee94781ae14

Runtime output

```
raised CONSTRAINT_ERROR : show_sat_op.adb:9 overflow check failed
```

Here, we're using the standard + operator, which raises a Constraint_Error exception in the C := A + B; statement due to an overflow. Let's now override the addition operator and enforce saturation when the result is out of range:

Listing 1103: fixed_point.ads

```

1 package Fixed_Point is
2
3     D : constant := 2.0 ** (-31);
4     type TQ31 is delta D range -1.0 .. 1.0 - D;
5
6     function "+" (Left, Right : TQ31)
7                 return TQ31;
8
9 end Fixed_Point;

```

Listing 1104: fixed_point.adb

```

1 package body Fixed_Point is
2

```

(continues on next page)

(continued from previous page)

```

3  function "+" (Left, Right : TQ31)
4      return TQ31
5  is
6      type TQ31_2 is
7          delta TQ31'Delta
8          range TQ31'First * 2.0 .. TQ31'Last * 2.0;
9
10     L : constant TQ31_2 := TQ31_2 (Left);
11     R : constant TQ31_2 := TQ31_2 (Right);
12     Res : TQ31_2;
13 begin
14     Res := L + R;
15
16     if Res > TQ31_2 (TQ31'Last) then
17         return TQ31'Last;
18     elsif Res < TQ31_2 (TQ31'First) then
19         return TQ31'First;
20     else
21         return TQ31 (Res);
22     end if;
23 end "+";
24
25 end Fixed_Point;

```

Listing 1105: show_sat_op.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Fixed_Point; use Fixed_Point;
3
4  procedure Show_Sat_Op is
5      A, B, C : TQ31;
6  begin
7      A := TQ31'Last;
8      B := TQ31'Last;
9      C := A + B;
10
11     Put_Line (A'Image & " + "
12              & B'Image & " = "
13              & C'Image);
14
15     A := TQ31'First;
16     B := TQ31'First;
17     C := A + B;
18
19     Put_Line (A'Image & " + "
20              & B'Image & " = "
21              & C'Image);
22
23 end Show_Sat_Op;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Operator_Overriding.Fixed_
 ↪ Point_Operator_Overloading
 MD5: 6317bcf9c278c01f86dbdcdb761d86240

Runtime output

```

0.9999999995 + 0.9999999995 = 0.9999999995
-1.0000000000 + -1.0000000000 = -1.0000000000

```

In the implementation of the overridden `+` operator of the `TQ31` type, we declare another type (`TQ31_2`) with a wider range than `TQ31`. We use variables of the `TQ31_2` type to perform the actual addition, and then we verify whether the result is still in `TQ31`'s range. If it is, we simply convert the result *back* to the `TQ31` type. Otherwise, we saturate it — using either the first or last value of the `TQ31` type.

When overriding operators, the overridden operator replaces the original one. For example, in the `A + B` operation of the `Show_Sat_Op` procedure above, we're using the overridden version of the `+` operator, which performs saturation. Therefore, this operation doesn't raise an exception (as it was the case with the original `+` operator).

26.3.7 Nonreturning procedures

Usually, when calling a procedure `P`, we expect that it returns to the caller's *thread of control* after performing some action in the body of `P`. However, there are situations where a procedure never returns. We can indicate this fact by using the `No_Return` aspect in the subprogram declaration.

A typical example is that of a server that is designed to run forever until the process is killed or the machine where the server runs is switched off. This server can be implemented as an endless loop. For example:

Listing 1106: `servers.ads`

```
1 package Servers is
2
3     procedure Run_Server
4         with No_Return;
5
6 end Servers;
```

Listing 1107: `servers.adb`

```
1 package body Servers is
2
3     procedure Run_Server is
4     begin
5         pragma Warnings
6             (Off,
7              "implied return after this statement");
8         while True loop
9             -- Processing happens here...
10            null;
11        end loop;
12    end Run_Server;
13
14 end Servers;
```

Listing 1108: `show_endless_loop.adb`

```
1 with Servers; use Servers;
2
3 procedure Show_Endless_Loop is
4 begin
5     Run_Server;
6 end Show_Endless_Loop;
```

Code block metadata

Project: `Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.`
 ↪ `Server_Proc`

(continues on next page)

(continued from previous page)

MD5: 3f859b6e2aca8e31367658632e84126c

In this example, `Run_Server` doesn't exit from the `while True` loop, so it never returns to the `Show_Endless_Loop` procedure.

The same situation happens when we call a procedure that raises an exception unconditionally. In that case, exception handling is triggered, so that the procedure never returns to the caller. An example is that of a logging procedure that writes a message before raising an exception internally:

Listing 1109: `loggers.ads`

```
1 package Loggers is
2
3     Logged_Failure : exception;
4
5     procedure Log_And_Raise (Msg : String)
6         with No_Return;
7
8 end Loggers;
```

Listing 1110: `loggers.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Loggers is
4
5     procedure Log_And_Raise (Msg : String) is
6     begin
7         Put_Line (Msg);
8         raise Logged_Failure;
9     end Log_And_Raise;
10
11 end Loggers;
```

Listing 1111: `show_no_return_exception.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Loggers; use Loggers;
3
4 procedure Show_No_Return_Exception is
5     Check_Passed : constant Boolean := False;
6 begin
7     if not Check_Passed then
8         Log_And_Raise ("Check failed!");
9         Put_Line ("This line will not be reached!");
10    end if;
11 end Show_No_Return_Exception;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.Log_Exception
MD5: 10b4933d8c862d14ade54935cbd2b541

In this example, `Log_And_Raise` writes a message to the user and raises the `Logged_Failure`, so it never returns to the `Show_No_Return_Exception` procedure.

We could implement exception handling in the `Show_No_Return_Exception` procedure, so that the `Logged_Failure` exception could be handled there after it's raised in `Log_And_Raise`. However, this wouldn't be considered a *normal* return to the procedure

because it wouldn't return to the point where it should (i.e. to the point where Put_Line is about to be called, right after the call to the Log_And_Raise procedure).

If a nonreturning procedure returns nevertheless, this is considered a program error, so that the Program_Error exception is raised. For example:

Listing 1112: loggers.ads

```

1 package Loggers is
2
3     Logged_Failure : exception;
4
5     procedure Log_And_Raise (Msg : String)
6         with No_Return;
7
8 end Loggers;
```

Listing 1113: loggers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Loggers is
4
5     procedure Log_And_Raise (Msg : String) is
6     begin
7         Put_Line (Msg);
8     end Log_And_Raise;
9
10 end Loggers;
```

Listing 1114: show_no_return_exception.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Loggers; use Loggers;
3
4 procedure Show_No_Return_Exception is
5     Check_Passed : constant Boolean := False;
6 begin
7     if not Check_Passed then
8         Log_And_Raise ("Check failed!");
9         Put_Line ("This line will not be reached!");
10    end if;
11 end Show_No_Return_Exception;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Nonreturning_Procedures.
↳ Erroneous_Log_Exception
MD5: e44fd8df0529dda5749e85b9e300a999

Build output

```

show_no_return_exception.adb:9:07: warning: unreachable code [enabled by default]
loggers.adb:7:07: warning: implied return after this statement will raise Program_
↳ Error [enabled by default]
loggers.adb:7:07: warning: procedure "Log_And_Raise" is marked as No_Return_
↳ [enabled by default]
```

Runtime output

Check failed!

(continues on next page)

(continued from previous page)

```
raised PROGRAM_ERROR : loggers.adb:7 implicit return with No_Return
```

Here, `Program_Error` is raised when `Log_And_Raise` returns to the `Show_No_Return_Exception` procedure.

In the Ada Reference Manual

- 6.5.1 Nonreturning Subprograms¹⁹⁷

26.3.8 Inline subprograms

Inlining¹⁹⁸ refers to a kind of optimization where the code of a subprogram is expanded at the point of the call in place of the call itself.

In modern compilers, inlining depends on the optimization level selected by the user. For example, if we select the higher optimization level, the compiler will perform automatic inlining aggressively.

In the GNAT toolchain

The highest optimization level (-O3) of GNAT performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (-O2). Therefore, the general recommendation is to not *just* select -O3 for the optimized version of an application, but instead compare it the optimized version built with -O2.

It's important to highlight that the inlining we're referring above happens automatically, so the decision about which subprogram is inlined depends entirely on the compiler. However, in some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the `Inline` aspect.

Let's look at this example:

Listing 1115: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is
4         array (Positive range <>) of Float;
5
6     function Average (Data : Float_Array)
7         return Float
8         with Inline;
9
10 end Float_Arrays;
```

Listing 1116: float_arrays.adb

```
1 package body Float_Arrays is
2
3     function Average (Data : Float_Array)
4         return Float
```

(continues on next page)

¹⁹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-6-5-1.html>

¹⁹⁸ https://en.wikipedia.org/wiki/Inline_expansion

(continued from previous page)

```

5  is
6    Total : Float := 0.0;
7  begin
8    for Value of Data loop
9      Total := Total + Value;
10   end loop;
11   return Total / Float (Data'Length);
12 end Average;
13
14 end Float_Arrays;

```

Listing 1117: compute_average.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Float_Arrays; use Float_Arrays;
4
5  procedure Compute_Average is
6    Values      : constant Float_Array :=
7                  (10.0, 11.0, 12.0, 13.0);
8    Average_Value : Float;
9  begin
10   Average_Value := Average (Values);
11   Put_Line ("Average = "
12             & Float'Image (Average_Value));
13 end Compute_Average;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Inline_Subprograms.Inlining_
 ↪Float_Arrays
 MD5: 246bc11e8a969d69873f416f583f450e

Runtime output

```
Average =  1.15000E+01
```

When compiling this example, the compiler will most probably inline `Average` in the `Compute_Average` procedure. Note, however, that the `Inline` aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

** In the GNAT toolchain**

In order to effectively use the `Inline` aspect, we need to set the optimization level to at least `-O1` and use the `-gnatn` switch, which instructs the compiler to take the `Inline` aspect into account.

In addition to the `Inline` aspect, in GNAT, we also have the (implementation-defined) `Inline_Always` aspect. In contrast to the former aspect, however, the `Inline_Always` aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of

this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the `Inline` aspect, there might be situations where a subprogram has the `Inline_Always` aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

Note that we can use the `Inline` aspect for generic subprograms as well. When we do this, we indicate to the compiler that we wish it inlines all instances of that generic subprogram.

In the Ada Reference Manual

- 6.3.2 Inline Expansion of Subprograms¹⁹⁹

26.3.9 Null Procedures

Null procedures are procedures that don't have any effect, as their body is empty. We declare a null procedure by simply writing `is null` in its declaration. For example:

Listing 1118: null_procs.ads

```
1 package Null_Procs is
2
3     procedure Do_Nothing (Msg : String) is null;
4
5 end Null_Procs;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: a8a801e6c71d8177db61e4aa131b8832

As expected, calling a null procedure doesn't have any effect. For example:

Listing 1119: show_null_proc.adb

```
1 with Null_Procs; use Null_Procs;
2
3 procedure Show_Null_Proc is
4 begin
5     Do_Nothing ("Hello");
6 end Show_Null_Proc;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: 274eed0b0952b9aa7e422933ece42d86

Null procedures are equivalent to implementing a procedure with a body that only contains `null`. Therefore, the `Do_Nothing` procedure above is equivalent to this:

Listing 1120: null_procs.ads

```
1 package Null_Procs is
2
3     procedure Do_Nothing (Msg : String);
```

(continues on next page)

¹⁹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-6-3-2.html>

(continued from previous page)

```

4
5 end Null_Procs;

```

Listing 1121: null_procs.adb

```

1 package body Null_Procs is
2
3     procedure Do_Nothing (Msg : String) is
4     begin
5         null;
6     end Do_Nothing;
7
8 end Null_Procs;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Null_Proc_1
MD5: d0c9dc9265ebbaa9603681182dee1d92

Null procedures and overriding

We can use null procedures as a way to simulate interfaces for non-tagged types — similar to what actual interfaces do for tagged types. For example, we may start by declaring a type and null procedures that operate on that type. For example, let's model a very simple API:

Listing 1122: simple_storage.ads

```

1 package Simple_Storage is
2
3     type Storage_Model is null record;
4
5     procedure Set (S : in out Storage_Model;
6                  V : String) is null;
7     procedure Display (S : Storage_Model) is null;
8
9 end Simple_Storage;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_Storage_Model
MD5: 553e78bc15dcec1302be4b5f484ac21f

Here, the API of the `Storage_Model` type consists of the `Set` and `Display` procedures. Naturally, we can use objects of the `Storage_Model` type in an application, but this won't have any effect:

Listing 1123: show_null_proc.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Simple_Storage; use Simple_Storage;
3
4 procedure Show_Null_Proc is
5     S : Storage_Model;
6 begin
7     Put_Line ("Setting 24...");
8     Set (S, "24");
9     Display (S);
10 end Show_Null_Proc;

```


Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_Storage_Model
MD5: 523b3e7239e683f2d879caa9139106ca

Runtime output

Setting 24...

By itself, the `Storage_Model` type is not very useful. However, we can derive other types from it and override the null procedures. Let's say we want to implement the `Integer_Storage` type to store an integer value:

Listing 1124: simple_storage.ads

```
1 package Simple_Storage is
2
3     type Storage_Model is null record;
4
5     procedure Set (S : in out Storage_Model;
6                   V :      String) is null;
7     procedure Display (S : Storage_Model) is null;
8
9     type Integer_Storage is private;
10
11     procedure Set (S : in out Integer_Storage;
12                  V :      String);
13     procedure Display (S : Integer_Storage);
14
15 private
16
17     type Integer_Storage is record
18         V : Integer := 0;
19     end record;
20
21 end Simple_Storage;
```

Listing 1125: simple_storage.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Storage is
4
5     procedure Set (S : in out Integer_Storage;
6                   V :      String) is
7     begin
8         S.V := Integer'Value (V);
9     end Set;
10
11     procedure Display (S : Integer_Storage) is
12     begin
13         Put_Line ("Value: " & S.V'Image);
14     end Display;
15
16 end Simple_Storage;
```

Listing 1126: show_null_proc.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Simple_Storage; use Simple_Storage;
3
```

(continues on next page)

(continued from previous page)

```

4 procedure Show_Null_Proc is
5   S : Integer_Storage;
6 begin
7   Put_Line ("Setting 24...");
8   Set (S, "24");
9   Display (S);
10 end Show_Null_Proc;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Subprograms.Null_Procedures.Simple_Storage_Model
 MD5: 55d491d1ef72fb7be2bf0d2a212f335b

Runtime output

Setting 24...
 Value: 24

In this example, we can view `Storage_Model` as a sort of interface for derived non-tagged types, while the derived types — such as `Integer_Storage` — provide the actual implementation.

The section on *null records* (page 450) contains an extended example that makes use of null procedures.

** In the Ada Reference Manual**

- [6.7 Null Procedures](#)²⁰⁰

26.4 Exceptions

26.4.1 Classification of Errors

When we talk about errors and erroneous behavior in Ada, we can classify them in one of the four categories:

- compilation errors — i.e. errors that an Ada compiler must detect at compilation time;
- runtime errors — i.e. errors that are detected by an Ada-based application using checks at runtime;
- bounded errors;
- erroneous execution.

In this section, we discuss each of these categories.

** In the Ada Reference Manual**

- [1.1.5 Classification of Errors](#)²⁰¹

²⁰⁰ <http://www.ada-auth.org/standards/22rm/html/RM-6-7.html>

²⁰¹ <http://www.ada-auth.org/standards/12rm/html/RM-1-1-5.html>

Compilation errors

In the category of compilation errors, the goal is to prevent compilers from accepting illegal programs. Here, any program that doesn't follow the rules described in the Ada Reference Manual is considered illegal. Those rules include not only simple syntax errors, but also more complicated semantic rules, such as the ones concerning *accessibility levels* (page 932) for access types.

Note that Ada — in contrast to many programming languages, which can be quite permissive — tries to prevent as many errors as possible at compilation time because of its focus on safety. However, even though a wide range of errors can be detected at compilation time, this doesn't mean that a legal Ada program is free of errors. Therefore, using methods such as static analysis or unit testing is important.

Runtime errors

When a rule cannot be verified at compilation time, a common strategy is to have the compiler insert runtime checks into the resulting application. We see details about these checks later on when we discuss *checks and exceptions* (page 808).

A typical example is an *overflow check* (page 813). Consider a calculation using variables: if this calculation leads to a result that isn't representable with the underlying data types, we cannot possibly store a value into a register or memory that can be considered correct — so we have to detect this situation. Unfortunately, because we're using variables, we obviously cannot verify the result of the calculation at compilation time, so we have to verify it at runtime.

As we've mentioned before, Ada strives for detecting as many erroneous conditions as possible, while other programming language would allow errors such as overflow errors to remain undetected — which would likely lead the application to misbehave. Those checks raise an exception if an erroneous condition is detected, so the programmer has the means — and the responsibility — to catch that exception and handle the situation properly (Note, however, that some of the runtime checks can be deactivated. We will discuss this topic later on.)

Bounded errors

For certain kinds of errors, the compiler might not be able to detect the error — neither at compilation time, nor with checks at runtime. Such errors are called bounded errors because their possible effects are *bounded*. In fact, the Ada Reference Manual describes each bounded error and its possible effects — one of those effects is raising the `Program_Error` exception.

Just as an example, consider the bounded error described in section 13.9.1 *Data Validity*²⁰², paragraphs 9:

If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an invalid representation. It is a bounded error to evaluate the value of such an object. If the error is detected, either `Constraint_Error` or `Program_Error` is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations.

Let's see a code example:

²⁰² <http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html>

Listing 1127: show_bounded_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Bounded_Error is
4   subtype Int_1_10 is
5     Integer range 1 .. 10;
6
7   I1      : Int_1_10;
8   I1_Overlay : Integer
9     with Address => I1'Address,
10      Import,
11      Volatile;
12 begin
13   I1_Overlay := 0;
14   -- ^^^^^^^^^^
15   -- We use this overlay to write an invalid
16   -- value to I1.
17
18   Put_Line ("I1 = " & I1'Image);
19   -- ^^^^^^^^^
20   -- Bounded error: value in
21   -- I1 is out of range.
22
23   I1 := I1 + 1;
24   -- ^^
25   -- Bounded error: using value
26   -- in operation that is out of
27   -- range.
28
29   Put_Line ("I1 = " & I1'Image);
30 end Show_Bounded_Error;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Classification_Of_Errors.
 ↪Data_Validity_Bounded_Error
 MD5: 770ebb7b6e0015e373e96c0dce250caa

Runtime output

```

I1 = 0
I1 = 1

```

In this example, we simulate a missing initialization by using an overlay (I1_Overlay). As a consequence, I1 has an invalid value that is out of the allowed range of the Int_1_10 subtype. This situation causes two bounded errors:

- a bounded error when I1 is evaluated in the call to Image; and
- a bounded error when the value of the right-sided I1 is evaluated — in the increment `I1 := I1 + 1`.

** In the Ada Reference Manual**

- 13.9.1 Data Validity²⁰³

²⁰³ <http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html>

Erroneous execution

Erroneous execution is similar to bounded errors in the sense that having the compiler detect the erroneous condition at compilation time or at runtime isn't possible. However, unlike bounded errors, the effects are usually nondeterministic: a bound on possible effects is not described by the language.

Again, as an example of erroneous execution, consider the description from section 13.9.1 [Data Validity](#)²⁰⁴, paragraph 12/3, which discusses the implications of using the `Unchecked_Conversion` function. Let's see a code example:

Listing 1128: `show_erroneous_execution.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Unchecked_Conversion;
3
4 procedure Show_Erroneous_Execution is
5   subtype Int_1_10 is
6     Integer range 1 .. 10;
7
8   function To_Int_1_10 is new
9     Ada.Unchecked_Conversion
10      (Source => Integer,
11       Target => Int_1_10);
12
13   I1 : Int_1_10 := To_Int_1_10 (0);
14   -- ~~~~~~
15   -- Bounded error
16 begin
17   Put_Line ("I1 = " & I1'Image);
18
19   I1 := I1 + 1;
20   -- ~~~~~~
21   -- Erroneous execution: using value
22   -- in operation that is out of range.
23
24   Put_Line ("I1 = " & I1'Image);
25 end Show_Erroneous_Execution;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Classification_Of_Errors.
↳ Data_Validity_Erroneous_Execution
MD5: 19218e9bb2e153366dea9114a5e59314

Build output

```
show_erroneous_execution.adb:8:04: warning: types for unchecked conversion have
↳ different sizes [-gnatwz]
```

Runtime output

```
I1 = 0
I1 = 1
```

It is considered to be a bounded error to use the `To_Int_1_10` function (based on `Unchecked_Conversion`) with a value that is invalid for the target data type. However, if we use the invalid value of `I1` in an operation such as the `I1 := I1 + 1` assignment, this leads to erroneous execution, and the effects are unpredictable: they aren't described in the Ada Reference Manual, as they are nondeterministic.

²⁰⁴ <http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html>

i In the Ada Reference Manual

- 13.9.1 Data Validity²⁰⁵

26.4.2 Asserts

When we want to indicate a condition in the code that must always be valid, we can use the pragma `Assert`. As the name implies, when we use this pragma, we're *asserting* some truth about the source-code. (We can also use the procedural form, as we'll see later.)

i Important

Another method to assert the truth about the source-code is to use pre and post-conditions.

A simple assert has this form:

Listing 1129: show_pragma_assert.adb

```

1 procedure Show_Pragma_Assert is
2   I : constant Integer := 10;
3
4   pragma Assert (I = 10);
5 begin
6   null;
7 end Show_Pragma_Assert;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Pragma_Assert_1
MD5: 8d40817304515169d0d5670904ca1e01

In this example, we're asserting that the value of `I` is always 10. We could also display a message if the assertion is false:

Listing 1130: show_pragma_assert.adb

```

1 procedure Show_Pragma_Assert is
2   I : constant Integer := 11;
3
4   pragma Assert (I = 10, "I is not 10");
5 begin
6   null;
7 end Show_Pragma_Assert;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Pragma_Assert_2
MD5: b70fa67c92542ade39c388964ce12302

Build output

```
show_pragma_assert.adb:4:19: warning: assertion will fail at run time [-gnatw.a]
```

Runtime output

²⁰⁵ <http://www.ada-auth.org/standards/12rm/html/RM-13-9-1.html>

```
raised ADA ASSERTIONS ASSERTION_ERROR : I is not 10
```

Similarly, we can use the procedural form of Assert. For example, the code above can be implemented as follows:

Listing 1131: show_procedure_assert.adb

```
1 with Ada.Assertions; use Ada.Assertions;
2
3 procedure Show_Procedure_Assert is
4   I : constant Integer := 11;
5
6 begin
7   Assert (I = 10, "I is not 10");
8 end Show_Procedure_Assert;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Procedure_Assert
MD5: cbab23645ff89d4adffcaaddaeb6f0e3

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : I is not 10
```

Note that a call to Assert is simply translated to a check — and the Assertion_Error exception from the Ada.Assertions package being raised in the case that the check fails. For example, the code above roughly corresponds to this:

Listing 1132: show_assertion_error.adb

```
1 with Ada.Assertions; use Ada.Assertions;
2
3 procedure Show_Assertion_Error is
4   I : constant Integer := 11;
5
6 begin
7   if I /= 10 then
8     raise Assertion_Error with "I is not 10";
9   end if;
10
11 end Show_Assertion_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Asserts.Assertion_Error
MD5: 9c846acf998ca7adabd47c3b5a6ce39f

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : I is not 10
```

In the Ada Reference Manual

- [11.4.2 Pragmas Assert and Assertion_Policy](#)²⁰⁶

²⁰⁶ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html>

26.4.3 Assertion policies

We can activate and deactivate assertions based on assertion policies. We can do that by using the pragma `Assertion_Policy`. As an argument to this pragma, we indicate whether a specific policy must be checked or ignored.

For example, we can deactivate assertion checks by specifying `Assert => Ignore`. Similarly, we can activate assertion checks by specifying `Assert => Check`. Let's see a code example:

Listing 1133: `show_pragma_assertion_policy.adb`

```

1 procedure Show_Pragma_Assertion_Policy is
2   I : constant Integer := 11;
3
4   pragma Assertion_Policy (Assert => Ignore);
5 begin
6   pragma Assert (I = 10);
7 end Show_Pragma_Assertion_Policy;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Pragma_Assertion_Policy_1
 MD5: 39b8aa4a34b6169c03b54074f4136519

Build output

```

show_pragma_assertion_policy.adb:6:19: warning: assertion would fail at run time [-gnatw.a]
```

Here, we're specifying that asserts shall be ignored. Therefore, the call to the pragma `Assert` doesn't raise an exception. If we replace `Ignore` with `Check` in the call to `Assertion_Policy`, the assert will raise the `Assertion_Error` exception.

The following table presents all policies that we can set:

Policy	Description
<code>Assert</code>	Check assertions
<code>Static_Predicate</code>	Check static predicates
<code>Dynamic_Predicate</code>	Check dynamic predicates
<code>Pre</code>	Check pre-conditions
<code>Pre'Class</code>	Check pre-condition of classes of tagged types
<code>Post</code>	Check post-conditions
<code>Post'Class</code>	Check post-condition of classes of tagged types
<code>Type_Invariant</code>	Check type invariants
<code>Type_Invariant'Class</code>	Check type invariant of classes of tagged types

In the GNAT toolchain

Compilers are free to include policies that go beyond the ones listed above. For example, GNAT includes the following policies — called *assertion kinds* in this context:

- `Assertions`
- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`

- Debug
- Ghost
- Initial_Condition
- Invariant
- Invariant'Class
- Loop_Invariant
- Loop_Variant
- Postcondition
- Precondition
- Predicate
- Refined_Post
- Statement_Assertions
- Subprogram_Variant

Also, in addition to Check and Ignore, GNAT allows you to set a policy to Disable and Suppressible.

You can read more about them in the [GNAT Reference Manual](#)²⁰⁷.

You can specify multiple policies in a single call to `Assertion_Policy`. For example, you can activate all policies by writing:

Listing 1134: `show_multiple_assertion_policies.adb`

```

1  procedure Show_Multiple_Assertion_Policies is
2      pragma Assertion_Policy
3      (Assert          => Check,
4       Static_Predicate => Check,
5       Dynamic_Predicate => Check,
6       Pre             => Check,
7       Pre'Class        => Check,
8       Post            => Check,
9       Post'Class       => Check,
10      Type_Invariant   => Check,
11      Type_Invariant'Class => Check);
12  begin
13      null;
14  end Show_Multiple_Assertion_Policies;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Multiple_Assertion_Policies
 MD5: 3abbbf97160b755b84cc4f7e652ca5551

In the GNAT toolchain

With GNAT, policies can be specified in multiple ways. In addition to calls to `Assertion_Policy`, you can use [configuration pragmas files](#)²⁰⁸. You can use these files to specify all pragmas that are relevant to your application, including `Assertion_Policy`. In addition, you can manage the granularity for those pragmas. For example, you can

²⁰⁷ https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Assertion_005fPolicy

use a global configuration pragmas file for your complete application, or even different files for each source-code file you have.

Also, by default, all policies listed in the table above are deactivated, i.e. they're all set to Ignore. You can use the command-line switch `-gnata` to activate them.

Note that the `Assert` procedure raises an exception independently of the assertion policy (`Assertion_Policy (Assert => Ignore)`). For example:

Listing 1135: `show_assert_procedure_policy.adb`

```

1 with Ada.Text_IO;    use Ada.Text_IO;
2 with Ada.Assertions; use Ada.Assertions;
3
4 procedure Show_Assert_Procedure_Policy is
5   pragma Assertion_Policy (Assert => Ignore);
6
7   I : constant Integer := 1;
8 begin
9   Put_Line ("----- Pragma Assert -----");
10  pragma Assert (I = 0);
11
12  Put_Line ("---- Procedure Assert ----");
13  Assert (I = 0);
14
15  Put_Line ("Finished.");
16 end Show_Assert_Procedure_Policy;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Assertion_Policies.Assert_Procedure_Policy
 MD5: 7be3bab24d856081afeddabe40afc84f

Build output

```
show_assert_procedure_policy.adb:10:19: warning: assertion would fail at run time
↳ [-gnatw.a]
```

Runtime output

```

----- Pragma Assert -----
---- Procedure Assert ----

raised ADA ASSERTIONS ASSERTION_ERROR : a-assert.adb:42
```

Here, the `pragma Assert` is ignored due to the assertion policy. However, the call to `Assert` is not ignored.

In the Ada Reference Manual

- 11.4.2 Pragmas `Assert` and `Assertion_Policy`²⁰⁹

²⁰⁸ https://gcc.gnu.org/onlinedocs/gnat_ugn/The-Configuration-Pragmas-Files#The-Configuration-Pragmas-Files

²⁰⁹ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-2.html>

26.4.4 Checks and exceptions

This table shows all language-defined checks and the associated exceptions:

Check	Exception
Access_Check	Constraint_Error
Discriminant_Check	Constraint_Error
Division_Check	Constraint_Error
Index_Check	Constraint_Error
Length_Check	Constraint_Error
Overflow_Check	Constraint_Error
Range_Check	Constraint_Error
Tag_Check	Constraint_Error
Accessibility_Check	Program_Error
Allocation_Check	Program_Error
Elaboration_Check	Program_Error
Program_Error_Check	Program_Error
Storage_Check	Storage_Error
Tasking_Check	Tasking_Error

In addition, we can use `All_Checks` to refer to all those checks above at once.

Let's discuss each check and see code examples where those checks are performed. Note that all examples are erroneous, so please avoid reusing them elsewhere.

Access Check

As you know, an object of an access type might be null. It would be an error to dereference this object, as it doesn't indicate a valid position in memory. Therefore, the access check verifies that an access object is not null when dereferencing it. For example:

Listing 1136: `show_access_check.adb`

```

1 procedure Show_Access_Check is
2
3   type Integer_Access is access Integer;
4
5   AI : Integer_Access;
6 begin
7   AI.all := 10;
8 end Show_Access_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Access_
 ↳ Check
 MD5: 4db8b63efb23caa7da926d4ec9f204bf

Build output

```

show_access_check.adb:5:04: warning: variable "AI" is read but never assigned [-  

↳ gnatwv]
show_access_check.adb:7:04: warning: null value not allowed here [enabled by_  

↳ default]
show_access_check.adb:7:04: warning: Constraint_Error will be raised at run time_  

↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_access_check.adb:7 access check failed
```

Here, the value of AI is null by default, so we cannot dereference it.

The access check also performs this verification when assigning to a subtype that excludes null (**not null access**). (You can find more information about this topic in the section about *not null access* (page 952).) For example:

Listing 1137: show_access_check.adb

```

1 procedure Show_Access_Check is
2
3   type Integer_Access is
4     access all Integer;
5
6   type Safe_Integer_Access is
7     not null access all Integer;
8
9   AI : Integer_Access;
10  SAI : Safe_Integer_Access := new Integer;
11
12 begin
13   SAI := Safe_Integer_Access (AI);
14 end Show_Access_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Access_↵
 ↵Check_2
 MD5: 47895a404e2a111476cd67f43c12d4b5

Build output

```

show_access_check.adb:9:04: warning: variable "AI" is read but never assigned [-
↵gnatwv]
show_access_check.adb:13:32: warning: null value not allowed here [enabled by ↵
↵default]
show_access_check.adb:13:32: warning: Constraint_Error will be raised at run time ↵
↵[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_access_check.adb:13 access check failed
```

Here, the value of AI is null (by default), so we cannot assign it to SAI because its type excludes null.

Note that, if we remove the `:= new Integer` assignment from the declaration of SAI, the null exclusion fails in the declaration itself (because the default value of the access type is **null**).

Discriminant Check

As we've seen earlier, a variant record is a record with discriminants that allows for changing its structure. In operations such as an assignment, it's important to ensure that the discriminants of the objects match — i.e. to ensure that the structure of the objects matches. The discriminant check verifies whether this is the case. For example:

Listing 1138: show_discriminant_check.adb

```

1 procedure Show_Discriminant_Check is
2
3     type Rec (Valid : Boolean) is record
4         case Valid is
5             when True =>
6                 Counter : Integer;
7             when False =>
8                 null;
9         end case;
10    end record;
11
12    R : Rec (Valid => False);
13 begin
14     R := (Valid => True,
15          Counter => 10);
16 end Show_Discriminant_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↳ Discriminant_Check
MD5: 665ab37962f8f9c129acac543b1eb15d

Build output

```

show_discriminant_check.adb:14:09: warning: incorrect value for discriminant "Valid
↳" [enabled by default]
show_discriminant_check.adb:14:09: warning: Constraint_Error will be raised at run_
↳time [enabled by default]
```

Runtime output

```

raised CONSTRAINT_ERROR : show_discriminant_check.adb:14 discriminant check failed
```

Here, R's discriminant (Valid) is **False**, so we cannot assign an object whose Valid discriminant is **True**.

Also, when accessing a component, the discriminant check ensures that this component exists for the current discriminant value:

Listing 1139: show_discriminant_check.adb

```

1 procedure Show_Discriminant_Check is
2
3     type Rec (Valid : Boolean) is record
4         case Valid is
5             when True =>
6                 Counter : Integer;
7             when False =>
8                 null;
9         end case;
10    end record;
11
12    R : Rec (Valid => False);
13    I : Integer;
14 begin
15     I := R.Counter;
16 end Show_Discriminant_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↪Discriminant_Check_2
 MD5: 440973b0be7c4261ddf3c2211a2c1325

Build output

```
show_discriminant_check.adb:15:10: warning: component not present in subtype of
↪"Rec" defined at line 12 [enabled by default]
show_discriminant_check.adb:15:10: warning: Constraint_Error will be raised at run_
↪time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_discriminant_check.adb:15 discriminant check failed
```

Here, R's discriminant (Valid) is **False**, so we cannot access the Counter component, for it only exists when the Valid discriminant is **True**.

Division Check

The division check verifies that we're not trying to divide a value by zero when using the **/**, **rem** and **mod** operators. For example:

Listing 1140: ops.ads

```
1 package Ops is
2   function Div_Op (A, B : Integer)
3     return Integer is
4     (A / B);
5
6   function Rem_Op (A, B : Integer)
7     return Integer is
8     (A rem B);
9
10  function Mod_Op (A, B : Integer)
11    return Integer is
12    (A mod B);
13 end Ops;
```

Listing 1141: show_division_check.adb

```
1 with Ops; use Ops;
2
3 procedure Show_Division_Check is
4   I : Integer;
5 begin
6   I := Div_Op (10, 0);
7   I := Rem_Op (10, 0);
8   I := Mod_Op (10, 0);
9 end Show_Division_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↪Division_Check
 MD5: 6ec0856be947eea6610cffaa0e875d45

Runtime output

```
raised CONSTRAINT_ERROR : ops.ads:4 divide by zero
```

All three calls in the `Show_Division_Check` procedure — to the `Div_Op`, `Rem_Op` and `Mod_Op` functions — can raise an exception because we're using 0 as the second argument, which makes the division check in those functions fail.

Index Check

We use indices to access components of an array. An index check verifies that the index we're using to access a specific component is within the array's bounds. For example:

Listing 1142: `show_index_check.adb`

```
1 procedure Show_Index_Check is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     function Value_Of (A : Integer_Array;
7                       I : Integer)
8         return Integer
9     is
10         type Half_Integer_Array is new
11             Integer_Array (A'First ..
12                           A'First + A'Length / 2);
13
14         A_2 : Half_Integer_Array := (others => 0);
15     begin
16         return A_2 (I);
17     end Value_Of;
18
19     Arr_1 : Integer_Array (1 .. 10) :=
20         (others => 1);
21
22 begin
23     Arr_1 (10) := Value_Of (Arr_1, 10);
24
25 end Show_Index_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Index_
↳ Check
MD5: fa791718701c4ac805badf368df9064e
```

Runtime output

```
raised CONSTRAINT_ERROR : show_index_check.adb:16 index check failed
```

The range of `A_2` — which is passed as an argument to the `Value_Of` function — is 1 to 6. However, in that function call, we're trying to access position 10, which is outside `A_2`'s bounds.

Length Check

In array assignments, both arrays must have the same length. To ensure that this is the case, a length check is performed. For example:

Listing 1143: show_length_check.adb

```

1 procedure Show_Length_Check is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   procedure Assign (To : out Integer_Array;
7                   From : Integer_Array) is
8   begin
9     To := From;
10  end Assign;
11
12  Arr_1 : Integer_Array (1 .. 10);
13  Arr_2 : Integer_Array (1 .. 9) :=
14    (others => 1);
15
16 begin
17   Assign (Arr_1, Arr_2);
18 end Show_Length_Check;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Length_
 ↳ Check
 MD5: a521afd0a46a67d260e8b0bd5f046ce4

Runtime output

raised CONSTRAINT_ERROR : show_length_check.adb:9 length check failed

Here, the length of Arr_1 is 10, while the length of Arr_2 is 9, so we cannot assign Arr_2 (From parameter) to Arr_1 (To parameter) in the Assign procedure.

Overflow Check

Operations on scalar objects might lead to overflow, which, if not checked, lead to wrong information being computed and stored. Therefore, an overflow check verifies that the value of a scalar object is within the base range of its type. For example:

Listing 1144: show_overflow_check.adb

```

1 procedure Show_Overflow_Check is
2   A, B : Integer;
3 begin
4   A := Integer'Last;
5   B := 1;
6
7   A := A + B;
8 end Show_Overflow_Check;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↳ Overflow_Check
 MD5: baa46d9085cbd14863aaa7e24dc7b9cc

Build output


```
show_overflow_check.adb:7:11: warning: value not in range of type "Standard.Integer"
↳ [enabled by default]
show_overflow_check.adb:7:11: warning: Constraint_Error will be raised at run time
↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_overflow_check.adb:7 overflow check failed
```

In this example, A already has the last possible value of the **Integer**'Base range, so increasing it by one causes an overflow error.

Range Check

The range check verifies that a scalar value is within a specific range — for instance, the range of a subtype. Let's see an example:

Listing 1145: show_range_check.adb

```
1 procedure Show_Range_Check is
2
3     subtype Int_1_10 is Integer range 1 .. 10;
4
5     I : Int_1_10;
6
7 begin
8     I := 11;
9 end Show_Range_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Range_
↳ Check
MD5: 54b1d67d98d97a58d4265a854fcfa992
```

Build output

```
show_range_check.adb:8:09: warning: value not in range of type "Int_1_10" defined
↳ at line 3 [enabled by default]
show_range_check.adb:8:09: warning: Constraint_Error will be raised at run time
↳ [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_range_check.adb:8 range check failed
```

In this example, we're trying to assign 11 to the variable I of the Int_1_10 subtype, which has a range from 1 to 10. Since 11 is outside that range, the range check fails.

Tag Check

The tag check ensures that the tag of a tagged object matches the expected tag in a dispatching operation. For example:

Listing 1146: p.ads

```
1 package P is
2
```

(continues on next page)

(continued from previous page)

```

3  type T is tagged null record;
4  type T1 is new T with null record;
5  type T2 is new T with null record;
6
7  end P;
```

Listing 1147: show_tag_check.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Tags;
3
4  with P;          use P;
5
6  procedure Show_Tag_Check is
7
8      A1 : T'Class := T1'(null record);
9      A2 : T'Class := T2'(null record);
10
11  begin
12      Put_Line ("A1'Tag: "
13              & Ada.Tags.Expanded_Name (A1'Tag));
14      Put_Line ("A2'Tag: "
15              & Ada.Tags.Expanded_Name (A2'Tag));
16
17      A2 := A1;
18
19  end Show_Tag_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.Tag_
Check
MD5: 5a685be7804200a884649f54c175ee42

Runtime output

```

A1'Tag: P.T1
A2'Tag: P.T2

raised CONSTRAINT_ERROR : show_tag_check.adb:17 tag check failed
```

Here, A1 and A2 have different tags:

- A1'Tag = T1'Tag, while
- A2'Tag = T2'Tag.

Since the tags don't match, the tag check fails in the assignment of A1 to A2.

Accessibility Check

The accessibility check verifies that the accessibility level of an entity matches the expected level. We discuss accessibility levels *in a later chapter* (page 932).

Let's look at an example that mixes access types and anonymous access types. Here, we use an anonymous access type in the declaration of A1 and a named access type in the declaration of A2:

Listing 1148: p.ads

```
1 package P is
2
3     type T is tagged null record;
4     type T_Class is access all T'Class;
5
6 end P;
```

Listing 1149: show_accessibility_check.adb

```
1 with P; use P;
2
3 procedure Show_Accessibility_Check is
4
5     A1 : access T'Class := new T;
6     A2 : T_Class;
7
8 begin
9     A2 := T_Class (A1);
10
11 end Show_Accessibility_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↳ Accessibility_Check
MD5: 7120d908b55ef576db93e9a15db257f2

Build output

```
show_accessibility_check.adb:9:19: warning: accessibility check fails [enabled by ↵  
↳ default]  
show_accessibility_check.adb:9:19: warning: Program_Error will be raised at run_↵  
↳ time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : show_accessibility_check.adb:9 accessibility check failed
```

The anonymous type (`access T'Class`), which is used in the declaration of A1, doesn't have the same accessibility level as the T_Class type. Therefore, the accessibility check fails during the T_Class (A1) conversion.

We can see the accessibility check failing in this example as well:

Listing 1150: show_accessibility_check.adb

```
1 with P; use P;
2
3 procedure Show_Accessibility_Check is
4
5     A : access T'Class := new T;
6
7     procedure P (A : T_Class) is null;
8
9 begin
10     P (T_Class (A));
11
12 end Show_Accessibility_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↪Accessibility_Check
 MD5: 97db82410dd3459249d0e7a97118b7ef

Build output

```
show_accessibility_check.adb:10:16: warning: accessibility check fails [enabled by ↪  

  ↪default]  

show_accessibility_check.adb:10:16: warning: Program_Error will be raised at run ↪  

  ↪time [enabled by default]
```

Runtime output

```
raised PROGRAM_ERROR : show_accessibility_check.adb:10 accessibility check failed
```

Again, the check fails in the T_Class (A) conversion and raises a Program_Error exception.

Allocation Check

The allocation check ensures, when a task is about to be created, that its master has not been completed. Also, it ensures that the finalization has not started.

This is an example adapted from [AI-00280](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00280.txt?rev=1.12&raw=N)²¹⁰:

Listing 1151: p.ads

```
1 with Ada.Finalization;  
2 with Ada.Unchecked_Deallocation;  
3  
4 package P is  
5   type T1 is new  
6     Ada.Finalization.Controlled with null record;  
7   procedure Finalize (X : in out T1);  
8  
9   type T2 is new  
10     Ada.Finalization.Controlled with null record;  
11   procedure Finalize (X : in out T2);  
12  
13   X1 : T1;  
14  
15   type T2_Ref is access T2;  
16   procedure Free is new  
17     Ada.Unchecked_Deallocation (T2, T2_Ref);  
18 end P;
```

Listing 1152: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 package body P is  
4  
5   procedure Finalize (X : in out T1) is  
6     X2 : T2_Ref := new T2;  
7   begin  
8     Put_Line ("Finalizing T1...");  
9     Free (X2);  
10  end Finalize;  
11
```

(continues on next page)

²¹⁰ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00280.txt?rev=1.12&raw=N>

(continued from previous page)

```
12  procedure Finalize (X : in out T2) is
13  begin
14      Put_Line ("Finalizing T2...");
15  end Finalize;
16
17  end P;
```

Listing 1153: show_allocation_check.adb

```
1  with P; use P;
2
3  procedure Show_Allocation_Check is
4      X2 : T2_Ref := new T2;
5  begin
6      Free (X2);
7  end Show_Allocation_Check;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
Allocation_Check
MD5: 915e8ab21e550c981503c014bcceade1

Runtime output

Finalizing T2...

raised PROGRAM_ERROR : finalize/adjust raised exception

Here, in the finalization of the X1 object of T1 type, we're trying to create an object of T2 type while the finalization of the master has already started. (Note that X1 was declared in the P package.) This is forbidden, so the allocation check raises a Program_Error exception.

Elaboration Check

The elaboration check verifies that subprograms — or protected entries, or task activations — have been elaborated before being called.

This is an example adapted from [AI-00064²¹¹](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00064.txt?rev=1.12&raw=N):

Listing 1154: p.ads

```
1  function P return Integer;
```

Listing 1155: p.adb

```
1  function P return Integer is
2  begin
3      return 1;
4  end P;
```

Listing 1156: show_elaboration_check.adb

```
1  with P;
2
3  procedure Show_Elaboration_Check is
4
5      function F return Integer;
```

(continues on next page)

²¹¹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00064.txt?rev=1.12&raw=N>

(continued from previous page)

```

6
7  type Pointer_To_Func is
8      access function return Integer;
9
10  X : constant Pointer_To_Func := P'Access;
11
12  Y : constant Integer := F;
13  Z : constant Pointer_To_Func := X;
14
15  -- Renaming-as-body
16  function F return Integer renames Z.all;
17  begin
18      null;
19  end Show_Elaboration_Check;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↳Elaboration_Check
MD5: 80a39df912aae8788296f81ee9d4a79e

Build output

```

show_elaboration_check.adb:12:28: warning: cannot call "F" before body seen
↳[enabled by default]
show_elaboration_check.adb:12:28: warning: Program_Error will be raised at run
↳time [enabled by default]

```

Runtime output

```

raised PROGRAM_ERROR : show_elaboration_check.adb:12 access before elaboration

```

This is a curious example: first, we declare a function `F` and assign the value returned by this function to constant `Y` in its declaration. Then, we declare `F` as a renamed function, thereby providing a body to `F` — this is called *renaming-as-body*. Consequently, the compiler doesn't complain that a body is missing for function `F`. (If you comment out the function renaming, you'll see that the compiler can then detect the missing body.) Therefore, at runtime, the elaboration check fails because the body of the first declaration of the `F` function is actually missing.

Program_Error_Check**Note**

This concept was introduced in Ada 2022.

As we've seen before, there are three checks that may raise a `Program_Error` exception: the `Accessibility_Check`, the `Allocation_Check` and the `Elaboration_Check`. In addition to that, we have the `Program_Error_Check`, which is actually a collection of various different checks that may raise a `Program_Error`, but don't have a category for themselves.

For completeness, these are the error conditions checked by the `Program_Error_Check` (listed in the [Action Item \(AI\) 12-0309 document](http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0309-1.txt?rev=1.5&raw=N)²¹²), according to their definition in the Ada Reference Manual:

²¹² <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0309-1.txt?rev=1.5&raw=N>

Ada Reference Manual	Paragraph	Description
3.2.4 Subtype Predicates ²¹³	(29.1	It checks that subtypes with predicates are not used to index an array in generic units.
5.5 Loop State-ments ²¹⁴	(8.1/!	It checks that the maximum number of chunks for statement-level parallelism is greater than zero.
6.4.1 Parameter Associations ²¹⁵	(13.4	It checks that the default value of an out parameter is convertible: an error occurs when we have an out parameter with Default_Value, and the actual is a view conversion of an unrelated type that does not have Default_Value.
12.5.1 Formal Private and Derived Types ²¹⁶	(23.3	It checks that there is no misuse of functions in a generic with a class-wide actual type.
13.3 Operational and Representation Attributes ²¹⁷	(75.1	It checks that there are no colliding External_Tag values.
B.3.3 Unchecked Union Types ²¹⁸	(22/2	It checks that there is no misuse of operations of Unchecked_Unions without inferable discriminants.

In the Ada Reference Manual

- 11.5 Suppressing Checks²¹⁹
- 3.2.4 Subtype Predicates²²⁰
- 5.5 Loop Statements²²¹
- 6.4.1 Parameter Associations²²²
- 12.5.1 Formal Private and Derived Types²²³
- 13.3 Operational and Representation Attributes²²⁴
- B.3.3 Unchecked Union Types²²⁵

²¹³ <http://www.ada-auth.org/standards/22rm/html/RM-3-2-4.html>

²¹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-5-5.html>

²¹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²¹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-12-5-1.html>

²¹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

²¹⁸ <http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html>

²¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-11-5.html>

²²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-2-4.html>

²²¹ <http://www.ada-auth.org/standards/22rm/html/RM-5-5.html>

²²² <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²²³ <http://www.ada-auth.org/standards/22rm/html/RM-12-5-1.html>

²²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

²²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-B-3-3.html>

Example of a Program_Error_Check

Just as an example, let's look at the check for subtype predicates in generic units:

Listing 1157: some_generic_package.ads

```

1 generic
2   type R is (<>);
3 package Some_Generic_Package is
4   procedure Process;
5 end Some_Generic_Package;
```

Listing 1158: some_generic_package.adb

```

1 package body Some_Generic_Package is
2
3   procedure Process is
4     type Arr is
5       array (R) of Integer;
6
7     Dummy : Arr := (others => 0);
8   begin
9     null;
10  end Process;
11
12 end Some_Generic_Package;
```

Listing 1159: show_subtype_predicate_programm_error.adb

```

1 with Some_Generic_Package;
2
3 procedure Show_Subtype_Predicate_Programm_Error is
4
5   type Custom_Range is range 1 .. 5
6   with Dynamic_Predicate =>
7     4 not in Custom_Range;
8
9   package P is new
10    Some_Generic_Package (Custom_Range);
11   use P;
12 begin
13   Process;
14 end Show_Subtype_Predicate_Programm_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↳ Subtype_Predicate_Programm_Error
 MD5: b1a5cc579393162dedecb6b65b75eef4

Build output

```

show_subtype_predicate_programm_error.adb:9:04: warning: in instantiation at some_
↳ generic_package.adb:5 [enabled by default]
show_subtype_predicate_programm_error.adb:9:04: warning: subtype "R" has predicate,
↳ not allowed as index subtype [enabled by default]
show_subtype_predicate_programm_error.adb:9:04: warning: Program_Error will be
↳ raised at run time [enabled by default]
```

Runtime output


```
raised PROGRAM_ERROR : some_generic_package.adb:5 improper use of generic subtype_
↳with predicate
```

Here, we're using the `Custom_Range` type for the formal type `R` in the instantiation of the generic package `Some_Generic_Package`. Since we use `R` as an index for the array type `Arr` (in the procedure `Process`), we cannot map a type to `R` that uses a predicate. Therefore, because `Custom_Range` type has a dynamic predicate, the `Program_Error` exception is raised.

Storage Check

The storage check ensures that the storage pool has enough space when allocating memory. Let's revisit an example that we *discussed earlier* (page 361):

Listing 1160: `custom_types.ads`

```
1 package Custom_Types is
2
3     type UInt_7 is range 0 .. 127;
4
5     type UInt_7_Reserved_Access is access UInt_7
6         with Storage_Size => 8;
7
8 end Custom_Types;
```

Listing 1161: `show_storage_check.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Custom_Types; use Custom_Types;
4
5 procedure Show_Storage_Check is
6
7     RAV1, RAV2 : UInt_7_Reserved_Access;
8
9 begin
10     Put_Line ("Allocating RAV1...");
11     RAV1 := new UInt_7;
12
13     Put_Line ("Allocating RAV2...");
14     RAV2 := new UInt_7;
15
16     New_Line;
17 end Show_Storage_Check;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
↳Storage_Check
MD5: 4e4bd284adb1c1d97f8f7563068c18de
```

Runtime output

```
Allocating RAV1...
Allocating RAV2...

raised STORAGE_ERROR : s-poosiz.adb:108 explicit raise
```

On each allocation (`new UInt_7`), a storage check is performed. Because there isn't enough reserved storage space before the second allocation, the checks fails and raises a `Storage_Error`.

age_Error exception.

Tasking_Check

The **Tasking_Check** ensures that all tasks have been activated successfully and that no terminated task is called. If the check fails, a **Tasking_Error** exception is raised.

Note

This concept was introduced in Ada 2022. It was created to group all checks that might raise the **Tasking_Error** exception.

Let's look at a simple example:

Listing 1162: workers.ads

```

1 package Workers is
2
3     task type Worker is
4         entry Start;
5     end Worker;
6
7 end Workers;
```

Listing 1163: workers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Workers is
4
5     task body Worker is
6     begin
7         Put_Line ("Task has started.");
8         delay 1.0;
9         Put_Line ("Task has finished.");
10    end Worker;
11
12 end Workers;
```

Listing 1164: show_tasking_check_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Workers; use Workers;
3
4 procedure Show_Tasking_Check_Error is
5     W : Worker;
6 begin
7     Put_Line ("W.Start...");
8     W.Start;
9     Put_Line ("Finished");
10 end Show_Tasking_Check_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Checks_And_Exceptions.
 ↪ Tasking_Check_Error
 MD5: 38f9093082d3fe545847ea3d22376e39

Build output

```
workers.adb:5:05: warning: no accept for entry "Start" [enabled by default]
```

Runtime output

```
Task has started.  
W.Start...  
Task has finished.  
  
raised TASKING_ERROR
```

In this example, the body of Worker doesn't have an **accept**. Therefore, no rendezvous can happen for the W.Start call. Since the task eventually terminates (as you can see in the user messages), the call to Start constitutes a call to a terminated task. This condition is checked by the **Tasking_Check**, which fails in this case, thereby raising a **Tasking_Error**.

26.4.5 Ada.Exceptions package

Note

Parts of this section were originally published as [Gem #142 : Exception-ally](#)²²⁶

The standard Ada run-time library provides the package `Ada.Exceptions`. This package provides a number of services to help analyze exceptions.

Each exception is associated with a (short) message that can be set by the code that raises the exception, as in the following code:

```
raise Constraint_Error with "some message";
```

Historically

Since Ada 2005, we can use the `raise Constraint_Error with "some message"` syntax. In Ada 95, you had to call the `Raise_Exception` procedure:

```
Ada.Exceptions.Raise_Exception      -- Ada 95  
(Constraint_Error'Identity, "some message");
```

In Ada 83, there was no way to do it at all.

The new syntax is now very convenient, and developers should be encouraged to provide as much information as possible along with the exception.

In the GNAT toolchain

The length of the message is limited to 200 characters by default in GNAT, and messages longer than that will be truncated.

In the Ada Reference Manual

- [11.4.1 The Package Exceptions](#)²²⁷

²²⁶ <https://www.adacore.com/gems/gem-142-exceptions>

²²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-11-4-1.html>

Retrieving exception information

Exceptions also embed information set by the run-time itself that can be retrieved by calling the `Exception_Information` function. The function `Exception_Information` also displays the `Exception_Message`.

For example:

```
exception
  when E : others =>
    Put_Line
      (Ada.Exceptions.Exception_Information (E));
```

In the GNAT toolchain

In the case of GNAT, the information provided by an exception might include the source location where the exception was raised and a nonsymbolic traceback.

You can also retrieve this information individually. Here, you can use:

- the `Exception_Name` functions — and its derivatives `Wide_Exception_Name` and `Wide_Wide_Exception_Name` — to retrieve the name of an exception.
- the `Exception_Message` function to retrieve the message associated with an exception.

Let's see a complete example:

Listing 1165: `show_exception_info.adb`

```
1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Ada.Exceptions; use Ada.Exceptions;
3
4  procedure Show_Exception_Info is
5
6      Custom_Exception : exception;
7
8      procedure Nested is
9      begin
10         raise Custom_Exception
11         with "We got a problem";
12     end Nested;
13
14     begin
15         Nested;
16
17     exception
18         when E : others =>
19             Put_Line ("Exception info: "
20                     & Exception_Information (E));
21             Put_Line ("Exception name: "
22                     & Exception_Name (E));
23             Put_Line ("Exception msg:  "
24                     & Exception_Message (E));
25     end Show_Exception_Info;
```

Collecting exceptions

Save_Occurrence

You can save an exception occurrence using the `Save_Occurrence` procedure. (Note that a `Save_Occurrence` function exists as well.)

For example, the following application collects exceptions into a list and displays them after running the `Test_Exceptions` procedure:

Listing 1166: exception_tests.ads

```
1 with Ada.Exceptions; use Ada.Exceptions;
2
3 package Exception_Tests is
4
5     Custom_Exception : exception;
6
7     type All_Exception_Occur is
8         array (Positive range <>) of
9             Exception_Occurrence;
10
11     procedure Test_Exceptions
12         (All_Occur : in out All_Exception_Occur;
13          Last_Occur : out Integer);
14
15 end Exception_Tests;
```

Listing 1167: exception_tests.adb

```
1 package body Exception_Tests is
2
3     procedure Save_To_List
4         (E      : Exception_Occurrence;
5          All_Occur : in out All_Exception_Occur;
6          Last_Occur : in out Integer)
7     is
8         L : Integer renames Last_Occur;
9         O : All_Exception_Occur renames All_Occur;
10    begin
11        L := L + 1;
12        if L > O'Last then
13            raise Constraint_Error
14                with "Cannot save occurrence";
15        end if;
16
17        Save_Occurrence (Target => O (L),
18                        Source => E);
19    end Save_To_List;
20
21    procedure Test_Exceptions
22        (All_Occur : in out All_Exception_Occur;
23         Last_Occur : out Integer)
24    is
25
26        procedure Nested_1 is
27        begin
28            raise Custom_Exception
29                with "We got a problem";
30        exception
31            when E : others =>
32                Save_To_List (E,
```

(continues on next page)

(continued from previous page)

```

33         All_Occur,
34         Last_Occur);
35     end Nested_1;
36
37     procedure Nested_2 is
38     begin
39         raise Constraint_Error
40         with "Constraint is not correct";
41     exception
42         when E : others =>
43             Save_To_List (E,
44                         All_Occur,
45                         Last_Occur);
46     end Nested_2;
47
48     begin
49         Last_Occur := 0;
50
51         Nested_1;
52         Nested_2;
53     end Test_Exceptions;
54
55 end Exception_Tests;

```

Listing 1168: show_exception_info.adb

```

1  with Ada.Text_IO;    use Ada.Text_IO;
2  with Ada.Exceptions; use Ada.Exceptions;
3
4  with Exception_Tests; use Exception_Tests;
5
6  procedure Show_Exception_Info is
7      L : Integer;
8      O : All_Exception_Occur (1 .. 10);
9  begin
10     Test_Exceptions (0, L);
11
12     for I in 0 'First .. L loop
13         Put_Line (Exception_Information (O (I)));
14     end loop;
15 end Show_Exception_Info;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Save_
 ↳ Occurrence
 MD5: da0cc5db7039e1458dbcf8be49db969d

Runtime output

```

raised EXCEPTION_TESTS.CUSTOM_EXCEPTION : We got a problem

raised CONSTRAINT_ERROR : Constraint is not correct

```

In the `Save_To_List` procedure of the `Exception_Tests` package, we call the `Save_Occurrence` procedure to store the exception occurrence to the `All_Occur` array. In the `Show_Exception_Info`, we display all the exception occurrences that we collected.

Read and Write attributes

Similarly, we can use files to read and write exception occurrences. To do that, we can simply use the Read and Write attributes.

Listing 1169: exception_occurrence_stream.adb

```
1  with Ada.Text_IO;
2
3  with Ada.Streams.Stream_IO;
4  use  Ada.Streams.Stream_IO;
5
6  with Ada.Exceptions;
7  use  Ada.Exceptions;
8
9  procedure Exception_Occurrence_Stream is
10
11     Custom_Exception : exception;
12
13     S : Stream_Access;
14
15     procedure Nested_1 is
16     begin
17         raise Custom_Exception
18             with "We got a problem";
19     exception
20         when E : others =>
21             Exception_Occurrence'Write (S, E);
22     end Nested_1;
23
24     procedure Nested_2 is
25     begin
26         raise Constraint_Error
27             with "Constraint is not correct";
28     exception
29         when E : others =>
30             Exception_Occurrence'Write (S, E);
31     end Nested_2;
32
33     F      : File_Type;
34     File_Name : constant String :=
35         "exceptions_file.bin";
36 begin
37     Create (F, Out_File, File_Name);
38     S := Stream (F);
39
40     Nested_1;
41     Nested_2;
42
43     Close (F);
44
45     Read_Exceptions : declare
46         E : Exception_Occurrence;
47     begin
48         Open (F, In_File, File_Name);
49         S := Stream (F);
50
51         while not End_Of_File (F) loop
52             Exception_Occurrence'Read (S, E);
53
54             Ada.Text_IO.Put_Line
55                 (Exception_Information (E));
```

(continues on next page)

(continued from previous page)

```

56     end loop;
57     Close (F);
58     end Read_Exceptions;
59
60 end Exception_Occurrence_Stream;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
↳Occurrence_Stream
MD5: 3d9f2bd9480aa6dcc250b249b9ef4870

```

Runtime output

```

raised EXCEPTION_OCCURRENCE_STREAM.CUSTOM_EXCEPTION : We got a problem

raised CONSTRAINT_ERROR : Constraint is not correct

```

In this example, we store the exceptions raised in the application in the *exceptions_file.bin* file. In the exception part of procedures *Nested_1* and *Nested_2*, we call *Exception_Occurrence'Write* to store an exception occurrence in the file. In the *Read_Exceptions* block, we read the exceptions from the file by calling *Exception_Occurrence'Read*.

Debugging exceptions in the GNAT toolchain

Here is a typical exception handler that catches all unexpected exceptions in the application:

Listing 1170: main.adb

```

1  with Ada.Exceptions;
2  with Ada.Text_IO;   use Ada.Text_IO;
3
4  procedure Main is
5
6      procedure Nested is
7      begin
8          raise Constraint_Error
9              with "some message";
10     end Nested;
11
12     begin
13         Nested;
14
15     exception
16         when E : others =>
17             Put_Line
18                 (Ada.Exceptions.Exception_Information (E));
19 end Main;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
↳Information
MD5: f95068ca90d79b92a7c2031322349153

```

Runtime output


```
raised CONSTRAINT_ERROR : some message
```

The output we get when running the application is not very informative. To get more information, we need to rerun the program in the debugger. To make the session more interesting though, we should add debug information in the executable, which means using the `-g` switch in the **gnatmake** command.

The session would look like the following (omitting some of the output from the debugger):

```
> rm *.o      # Cleanup previous compilation
> gnatmake -g main.adb
> gdb ./main
(gdb) catch exception
(gdb) run
Catchpoint 1, CONSTRAINT_ERROR at 0x0000000000402860 in main.nested () at main.
↳adb:8
8          raise Constraint_Error with "some message";

(gdb) bt
#0  <__gnat_debug_raise_exception> (e=0x62ec40 <constraint_error>) at s-excdeb.
↳adb:43
#1  0x000000000040426f in ada.exceptions.complete_occurrence (x=x@entry=0x637050)
    at a-except.adb:934
#2  0x000000000040427b in ada.exceptions.complete_and_propagate_occurrence (
    x=x@entry=0x637050) at a-except.adb:943
#3  0x00000000004042d0 in <__gnat_raise_exception> (e=0x62ec40 <constraint_error>,
    message=...) at a-except.adb:982
#4  0x0000000000402860 in main.nested ()
#5  0x000000000040287c in main ()
```

And we now know exactly where the exception was raised. But in fact, we could have this information directly when running the application. For this, we need to bind the application with the switch `-E`, which tells the binder to store exception tracebacks in exception occurrences. Let's recompile and rerun the application.

```
> rm *.o      # Cleanup previous compilation
> gnatmake -g main.adb -bargs -E
> ./main
```

```
Exception name: CONSTRAINT_ERROR
Message: some message
Call stack traceback locations:
0x10b7e24d1 0x10b7e24ee 0x10b7e2472
```

The traceback, as is, is not very useful. We now need to use another tool that is bundled with GNAT, called **addr2line**. Here is an example of its use:

```
> addr2line -e main --functions --demangle 0x10b7e24d1 0x10b7e24ee 0x10b7e2472
/path/main.adb:8
_ada_main
/path/main.adb:12
main
/path/b~main.adb:240
```

This time we do have a symbolic backtrace, which shows information similar to what we got in the debugger.

For users on OSX machines, **addr2line** does not exist. On these machines, however, an equivalent solution exists. You need to link your application with an additional switch, and then use the tool **atos**, as in:

```

> rm *.o
> gnatmake -g main.adb -bargs -E -largS -Wl,-no_pie
> ./main

Exception name: CONSTRAINT_ERROR
Message: some message
Call stack traceback locations:
0x1000014d1 0x1000014ee 0x100001472
> atos -o main 0x1000014d1 0x1000014ee 0x100001472
main_nested.2550 (in main) (main.adb:8)
_ada_main (in main) (main.adb:12)
main (in main) + 90

```

We will now discuss a relatively new switch of the compiler, namely `-gnateE`. When used, this switch will generate extra information in exception messages.

Let's amend our test program to:

Listing 1171: main.adb

```

1 with Ada.Exceptions;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Nested (Index : Integer) is
7         type T_Array is array (1 .. 2) of Integer;
8         T : constant T_Array := (10, 20);
9     begin
10         Put_Line (T (Index)'Img);
11     end Nested;
12
13 begin
14     Nested (3);
15
16 exception
17     when E : others =>
18         Put_Line
19             (Ada.Exceptions.Exception_Information (E));
20 end Main;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exceptions_Package.Exception_
Information
MD5: 3590f2bf48f6ed1cf7745d576924cad4

```

Runtime output

```

raised CONSTRAINT_ERROR : main.adb:10:17 index check failed
index 3 not in 1..2

```

When running the application, we see that the exception information (traceback) is the same as before, but this time the exception message is set automatically by the compiler. So we know we got a `Constraint_Error` because an incorrect index was used at the named source location (main.adb, line 10). But the significant addition is the second line of the message, which indicates exactly the cause of the error. Here, we wanted to get the element at index 3, in an array whose range of valid indexes is from 1 to 2. (No need for a debugger in this case.)

The column information on the first line of the exception message is also very useful when dealing with null pointers. For instance, a line such as:

```
A := Rec1.Rec2.Rec3.Rec4.all;
```

where each of the Rec is itself a pointer, might raise `Constraint_Error` with a message "access check failed". This indicates for sure that one of the pointers is null, and by using the column information it is generally easy to find out which one it is.

26.4.6 Exception renaming

We can rename exceptions by using the an exception renaming declaration in this form `Renamed_Exception : exception renames Existing_Exception;`. For example:

Listing 1172: `show_exception_renaming.adb`

```
1 procedure Show_Exception_Renaming is
2   CE : exception renames Constraint_Error;
3 begin
4   raise CE;
5 end Show_Exception_Renaming;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exception_Renaming.Exception_↵Renaming
MD5: ff20825162ee9eef6ac8ed329da2a80f

Runtime output

```
raised CONSTRAINT_ERROR : show_exception_renaming.adb:4
```

Exception renaming creates a new view of the original exception. If we rename an exception from package A in package B, that exception will become visible in package B. For example:

Listing 1173: `internal_exceptions.ads`

```
1 package Internal_Exceptions is
2
3   Int_E : exception;
4
5 end Internal_Exceptions;
```

Listing 1174: `test_constraints.ads`

```
1 with Internal_Exceptions;
2
3 package Test_Constraints is
4
5   Ext_E : exception renames
6           Internal_Exceptions.Int_E;
7
8 end Test_Constraints;
```

Listing 1175: `show_exception_renaming_view.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 with Test_Constraints; use Test_Constraints;
5
6 procedure Show_Exception_Renaming_View is
```

(continues on next page)

(continued from previous page)

```

7 begin
8   raise Ext_E;
9 exception
10  when E : others =>
11    Put_Line
12    (Ada.Exceptions.Exception_Information (E));
13 end Show_Exception_Renaming_View;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Exception_Renaming.Exception_Renaming_View
 MD5: a44e2698170c6fab79241d0f33ef8c2e

Runtime output

```
raised INTERNAL_EXCEPTIONS.INT_E : show_exception_renaming_view.adb:8
```

Here, we're renaming the Int_E exception in the Test_Constraints package. The Int_E exception isn't directly visible in the Show_Exception_Renaming procedure because we're not **with**ing the Internal_Exceptions package. However, it is indirectly visible in that procedure via the renaming (Ext_E) in the Test_Constraints package.

***i* In the Ada Reference Manual**

- 8.5.2 Exception Renaming Declarations²²⁸

26.4.7 Out and Uninitialized***i* Note**

This section was originally written by Robert Dewar and published as [Gem #150: Out and Uninitialized](#)²²⁹

Perhaps surprisingly, the Ada standard indicates cases where objects passed to **out** and **in out** parameters might not be updated when a procedure terminates due to an exception. Let's take an example:

Listing 1176: show_out_uninitialized.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure Show_Out_Uninitialized is
3
4   procedure Local (A      : in out Integer;
5                   Error : Boolean) is
6   begin
7     A := 1;
8
9     if Error then
10      raise Program_Error;
11    end if;
12  end Local;
13

```

(continues on next page)

²²⁸ <http://www.ada-auth.org/standards/22rm/html/RM-8-5-2.html>

²²⁹ <https://www.adacore.com/gems/gem-150out-and-uninitialized>

(continued from previous page)

```

14   B : Integer := 0;
15
16   begin
17     Local (B, Error => True);
18   exception
19     when Program_Error =>
20       Put_Line ("Value for B is"
21                 & Integer'Image (B)); -- "0"
22   end Show_Out_Uninitialized;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↳Uninitialized_1
MD5: cebcf14e9fd088e38b98a5132d9fd998

Runtime output

Value for B is 0

This program outputs a value of 0 for B, whereas the code indicates that A is assigned before raising the exception, and so the reader might expect B to also be updated.

The catch, though, is that a compiler must by default pass objects of elementary types (scalars and access types) by copy and might choose to do so for other types (records, for example), including when passing **out** and **in out** parameters. So what happens is that while the formal parameter A is properly initialized, the exception is raised before the new value of A has been copied back into B (the copy will only happen on a normal return).

i In the GNAT toolchain

In general, any code that reads the actual object passed to an **out** or **in out** parameter after an exception is suspect and should be avoided. GNAT has useful warnings here, so that if we simplify the above code to:

Listing 1177: show_out_uninitialized_warnings.adb

```

1   with Ada.Text_IO; use Ada.Text_IO;
2
3   procedure Show_Out_Uninitialized_Warnings is
4
5     procedure Local (A : in out Integer) is
6     begin
7       A := 1;
8       raise Program_Error;
9     end Local;
10
11    B : Integer := 0;
12
13    begin
14      Local (B);
15    exception
16      when others =>
17        Put_Line ("Value for B is"
18                  & Integer'Image (B));
19    end Show_Out_Uninitialized_Warnings;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↳Uninitialized_2
MD5: 5b6960974c729ea37a70fb313d6e5084

Build output

```
show_out_uninitialized_warnings.adb:7:10: warning: assignment to pass-by-copy
↳ formal may have no effect [enabled by default]
show_out_uninitialized_warnings.adb:7:10: warning: "raise" statement may result
↳ in abnormal return (RM 6.4.1(17)) [enabled by default]
```

Runtime output

```
Value for B is 0
```

We now get a compilation warning that the pass-by-copy formal may have no effect.

Of course, GNAT is not able to point out all such errors (see first example above), which in general would require full flow analysis.

The behavior is different when using parameter types that the standard mandates be passed by reference, such as tagged types for instance. So the following code will work as expected, updating the actual parameter despite the exception:

Listing 1178: show_out_initialized_rec.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Out_Initialized_Rec is
4
5     type Rec is tagged record
6         Field : Integer;
7     end record;
8
9     procedure Local (A : in out Rec) is
10     begin
11         A.Field := 1;
12         raise Program_Error;
13     end Local;
14
15     V : Rec;
16
17 begin
18     V.Field := 0;
19     Local (V);
20 exception
21     when others =>
22         Put_Line ("Value of Field is"
23                 & V.Field'Img); -- "1"
24 end Show_Out_Initialized_Rec;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_
↳ Uninitialized_3
MD5: 370031a404657ea18ffabf3c1d507cd4
```

Runtime output

```
Value of Field is 1
```

** In the GNAT toolchain**

It's worth mentioning that GNAT provides a pragma called `Export_Procedure` that forces reference semantics on **out** parameters. Use of this pragma would ensure updates of the

actual parameter prior to abnormal completion of the procedure. However, this pragma only applies to library-level procedures, so the examples above have to be rewritten to avoid the use of a nested procedure, and really this pragma is intended mainly for use in interfacing with foreign code. The code below shows an example that ensures that B is set to 1 after the call to Local:

Listing 1179: exported_procedures.ads

```

1 package Exported_Procedures is
2
3   procedure Local (A      : in out Integer;
4                   Error : Boolean);
5   pragma Export_Procedure
6     (Local,
7      Mechanism => (A => Reference));
8
9 end Exported_Procedures;
```

Listing 1180: exported_procedures.adb

```

1 package body Exported_Procedures is
2
3   procedure Local (A      : in out Integer;
4                   Error : Boolean) is
5   begin A := 1;
6     if Error then
7       raise Program_Error;
8     end if;
9   end Local;
10
11 end Exported_Procedures;
```

Listing 1181: show_out_reference.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Exported_Procedures;
4 use Exported_Procedures;
5
6 procedure Show_Out_Reference is
7   B : Integer := 0;
8 begin
9   Local (B, Error => True);
10 exception
11   when Program_Error =>
12     Put_Line ("Value for B is"
13              & Integer'Image (B)); -- "1"
14 end Show_Out_Reference;
```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Out_Uninitialized.Out_

Uninitialized_4
MD5: aed2788be2b3ceec19b28421c53fc66

Runtime output

Value for B is 1

In the case of direct assignments to global variables, the behavior in the presence of exceptions is somewhat different. For predefined exceptions, most notably `Constraint_Error`, the optimization permissions allow some flexibility in whether a global variable is or is not updated when an exception occurs (see [Ada RM 11.6](http://www.ada-auth.org/standards/22rm/html/RM-11-6.html)²³⁰). For instance, the following code

²³⁰ <http://www.ada-auth.org/standards/22rm/html/RM-11-6.html>

makes an incorrect assumption:

```
X := 0;      -- about to try addition
Y := Y + 1; -- see if addition raises exception
X := 1      -- addition succeeded
```

A program is not justified in assuming that `X = 0` if the addition raises an exception (assuming `X` is a global here). So any such assumptions in a program are incorrect code which should be fixed.

i In the Ada Reference Manual

- 11.6 Exceptions and Optimization²³¹

26.4.8 Suppressing checks

pragma Suppress

i Note

This section was originally written by Gary Dismukes and published as [Gem #63: The Effect of Pragma Suppress](#)²³².

One of Ada's key strengths has always been its strong typing. The language imposes stringent checking of type and subtype properties to help prevent accidental violations of the type system that are a common source of program bugs in other less-strict languages such as C. This is done using a combination of compile-time restrictions (legality rules), that prohibit mixing values of different types, together with run-time checks to catch violations of various dynamic properties. Examples are checking values against subtype constraints and preventing dereferences of null access values.

At the same time, Ada does provide certain "loophole" features, such as `Unchecked_Conversion`, that allow selective bypassing of the normal safety features, which is sometimes necessary when interfacing with hardware or code written in other languages.

Ada also permits explicit suppression of the run-time checks that are there to ensure that various properties of objects are not violated. This suppression can be done using **pragma Suppress**, as well as by using a compile-time switch on most implementations — in the case of GNAT, with the `-gnatp` switch.

In addition to allowing all checks to be suppressed, **pragma Suppress** supports suppression of specific forms of check, such as `Index_Check` for array indexing, `Range_Check` for scalar bounds checking, and `Access_Check` for dereferencing of access values. (See section 11.5 of the Ada Reference Manual for further details.)

Here's a simple example of suppressing index checks within a specific subprogram:

```
procedure Main is
  procedure Sort_Array (A : in out Some_Array) is
    pragma Suppress (Index_Check);
    --      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    --      eliminate check overhead
  begin
    ...
```

(continues on next page)

²³¹ <http://www.ada-auth.org/standards/22rm/html/RM-11-6.html>

²³² <https://www.adacore.com/gems/gem-63>

(continued from previous page)

```

end Sort_Array;
end Main;

```

Unlike a feature such as `Unchecked_Conversion`, however, the purpose of check suppression is not to enable programs to subvert the type system, though many programmers seem to have that misconception.

What's important to understand about `pragma Suppress` is that it only gives permission to the implementation to remove checks, but doesn't require such elimination. The intention of `Suppress` is not to allow bypassing of Ada semantics, but rather to improve efficiency, and the Ada Reference Manual has a clear statement to that effect in the note in RM-11.5, paragraph 29:

There is no guarantee that a suppressed check is actually removed; hence a `pragma Suppress` should be used only for efficiency reasons.

There is associated Implementation Advice that recommends that implementations should minimize the code executed for checks that have been suppressed, but it's still the responsibility of the programmer to ensure that the correct functioning of the program doesn't depend on checks not being performed.

There are various reasons why a compiler might choose not to remove a check. On some hardware, certain checks may be essentially free, such as null pointer checks or arithmetic overflow, and it might be impractical or add extra cost to suppress the check. Another example where it wouldn't make sense to remove checks is for an operation implemented by a call to a run-time routine, where the check might be only a small part of a more expensive operation done out of line.

Furthermore, in many cases GNAT can determine at compile time that a given run-time check is guaranteed to be violated. In such situations, it gives a warning that an exception will be raised, and generates code specifically to raise the exception. Here's an example:

```

X : Integer range 1..10 := ...;

..

if A > B then
  X := X + 1;
..
end if;

```

For the assignment incrementing `X`, the compiler will normally generate machine code equivalent to:

```

Temp := X + 1;
if Temp > 10 then
  raise Constraint_Error;
end if;
X := Temp;

```

If range checks are suppressed, then the compiler can just generate the increment and assignment. However, if the compiler is able to somehow prove that `X = 10` at this point, it will issue a warning, and replace the entire assignment with simply:

```

raise Constraint_Error;

```

even though checks are suppressed. This is appropriate, because

1. we don't care about the efficiency of buggy code, and
2. there is no "extra" cost to the check, because if we reach that point, the code will unconditionally fail.

One other important thing to note about checks and `pragma Suppress` is this statement in the Ada RM (RM-11.5, paragraph 26):

If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.

In Ada, erroneous execution is a bad situation to be in, because it means that the execution of your program could have arbitrary nasty effects, such as unintended overwriting of memory. Note also that a program whose "correct" execution somehow depends on a given check being suppressed might work as the programmer expects, but could still fail when compiled with a different compiler, or for a different target, or even with a newer version of the same compiler. Other changes such as switching on optimization or making a change to a totally unrelated part of the code could also cause the code to start failing.

So it's definitely not wise to write code that relies on checks being removed. In fact, it really only makes sense to suppress checks once there's good reason to believe that the checks can't fail, as a result of testing or other analysis. Otherwise, you're removing an important safety feature of Ada that's intended to help catch bugs.

`pragma Unsuppress`

We can use `pragma Unsuppress` to reverse the effect of a `pragma Suppress`. While `pragma Suppress` gives permission to the compiler to remove a specific check, `pragma Unsuppress` revokes that permission.

Let's see an example:

Listing 1182: show_index_check.adb

```

1  procedure Show_Index_Check is
2
3      type Integer_Array is
4          array (Positive range <>) of Integer;
5
6      pragma Suppress (Index_Check);
7      -- from now on, the compiler may
8      -- eliminate index checks...
9
10     function Unchecked_Value_Of
11         (A : Integer_Array;
12          I : Integer)
13         return Integer
14     is
15         type Half_Integer_Array is new
16             Integer_Array (A'First ..
17                             A'First + A'Length / 2);
18
19         A_2 : Half_Integer_Array := (others => 0);
20     begin
21         return A_2 (I);
22     end Unchecked_Value_Of;
23
24     pragma Unsuppress (Index_Check);
25     -- from now on, index checks are
26     -- typically performed...
27
28     function Value_Of
29         (A : Integer_Array;
30          I : Integer)
31         return Integer
32     is
33         type Half_Integer_Array is new

```

(continues on next page)

(continued from previous page)

```

34     Integer_Array (A'First ..
35                   A'First + A'Length / 2);
36
37     A_2 : Half_Integer_Array := (others => 0);
38   begin
39     return A_2 (I);
40   end Value_Of;
41
42   Arr_1 : Integer_Array (1 .. 10) :=
43     (others => 1);
44
45   begin
46     Arr_1 (10) := Unchecked_Value_Of (Arr_1, 10);
47     Arr_1 (10) := Value_Of (Arr_1, 10);
48
49   end Show_Index_Check;

```

Code block metadata

Project: Courses.Advanced_Ada.Control_Flow.Exceptions.Pragma_Unsuppress.Pragma_↵Unsuppress
 MD5: 0585b78fd57913d3172c7ab1ea6f4864

Runtime output

```
raised CONSTRAINT_ERROR : show_index_check.adb:39 index check failed
```

In this example, we first use a **pragma Suppress** (Index_Check), so the compiler is allowed to remove the index check from the Unchecked_Value_Of function. (Therefore, depending on the compiler, the call to the Unchecked_Value_Of function may complete without raising an exception.) Of course, in this specific example, suppressing the index check masks a severe issue.

In contrast, an index check is performed in the Value_Of function because of the **pragma Unsuppress**. As a result, the index checks fails in the call to this function, which raises a Constraint_Error exception.

In the Ada Reference Manual

- 11.5 Suppressing Checks²³³

²³³ <http://www.ada-auth.org/standards/22rm/html/RM-11-5.html>

MODULAR PROGRAMMING

27.1 Packages

27.1.1 Package renaming

We've seen in the Introduction to Ada course that we can *rename packages* (page 45).

i In the Ada Reference Manual

- 10.1.1 Compilation Units - Library Units²³⁴

Grouping packages

A use-case that we haven't mentioned in that course is that we can apply package renaming to group individual packages into a common hierarchy. For example:

Listing 1183: driver_m1.ads

```
1 package Driver_M1 is
2
3 end Driver_M1;
```

Listing 1184: driver_m2.ads

```
1 package Driver_M2 is
2
3 end Driver_M2;
```

Listing 1185: drivers.ads

```
1 package Drivers
2   with Pure is
3
4 end Drivers;
```

Listing 1186: drivers-m1.ads

```
1 with Driver_M1;
2
3 package Drivers.M1 renames Driver_M1;
```

²³⁴ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

Listing 1187: drivers-m2.ads

```

1 with Driver_M2;
2
3 package Drivers.M2 renames Driver_M2;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_Renaming_1
 MD5: 8d6a6bec32f7ec4397de1faf9f0b44d9

Here, we're renaming the Driver_M1 and Driver_M2 packages as child packages of the Drivers package, which is a pure package.

i Important

Note that a package that is renamed as a child package cannot refer to information from its (non-renamed) parent. In other words, Driver_M1 (renamed as Drivers.M1) cannot refer to information from the Drivers package. For example:

Listing 1188: driver_m1.ads

```

1 package Driver_M1 is
2
3     Counter_2 : Integer := Drivers.Counter;
4
5 end Driver_M1;

```

Listing 1189: drivers.ads

```

1 package Drivers is
2
3     Counter : Integer := 0;
4
5 end Drivers;

```

Listing 1190: drivers-m1.ads

```

1 with Driver_M1;
2
3 package Drivers.M1 renames Driver_M1;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_Renaming_1_Refer_To_Parent
 MD5: d174746d8151d9a2cd048ad44e853850

Build output

```

driver_m1.ads:3:27: error: "Drivers" is undefined
gprbuild: *** compilation phase failed

```

As expected, compilation fails here because Drivers.Counter isn't visible in Driver_M1, even though the renaming (Drivers.M1) creates a virtual hierarchy.

Child of renamed package

Note that we cannot create a child package using a parent package name that was introduced by a renaming. For example, let's say we want to create a child package `Ext` for the `Drivers.M1` package we've seen earlier. We cannot just declare a `Drivers.M1.Ext` package like this:

```
package Drivers.M1.Ext is
end Drivers.M1.Ext;
```

because the parent unit cannot be a renaming. The solution is to actually extend the original (non-renamed) package:

Listing 1191: driver_m1-ext.ads

```
1 package Driver_M1.Ext is
2
3 end Driver_M1.Ext;
```

Listing 1192: dummy.adb

```
1 -- A package called Drivers.M1.Ext is
2 -- automatically available!
3
4 with Drivers.M1.Ext;
5
6 procedure Dummy is
7 begin
8     null;
9 end Dummy;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_
↳Renaming_1
MD5: e338d668dbd98b1a3917a8d3d948a439
```

This works fine because any child package of a package `P` is also a child package of a renamed version of `P`. (Therefore, because `Ext` is a child package of `Driver_M1`, it is also a child package of the renamed `Drivers.M1` package.)

Backwards-compatibility via renaming

We can also use renaming to ensure backwards-compatibility when changing the package hierarchy. For example, we could adapt the previous source-code by:

- converting `Driver_M1` and `Driver_M2` to child packages of `Drivers`, and
- using package renaming to *mimic* the original names (`Driver_M1` and `Driver_M2`).

This is the adapted code:

Listing 1193: drivers.ads

```
1 package Drivers
2     with Pure is
3
4 end Drivers;
```

Listing 1194: drivers-m1.ads

```
1  -- We've converted Driver_M1 to
2  -- Drivers.M1:
3
4  package Drivers.M1 is
5
6  end Drivers.M1;
```

Listing 1195: drivers-m2.ads

```
1  -- We've converted Driver_M2 to
2  -- Drivers.M2:
3
4  package Drivers.M2 is
5
6  end Drivers.M2;
```

Listing 1196: driver_m1.ads

```
1  -- Original Driver_M1 package still
2  -- available via package renaming:
3
4  with Drivers.M1;
5
6  package Driver_M1 renames Drivers.M1;
```

Listing 1197: driver_m2.ads

```
1  -- Original Driver_M2 package still
2  -- available via package renaming:
3
4  with Drivers.M2;
5
6  package Driver_M2 renames Drivers.M2;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Package_Renaming.Package_
↳Renaming_2
MD5: 27f8066b5f5954514fea51b6e9b9de81
```

Now, M1 and M2 are *actual* child packages of Drivers, but their original names are still available. By doing so, we ensure that existing software that makes use of the original packages doesn't break.

27.1.2 Private packages

In this section, we discuss the concept of private packages. However, before we proceed with the discussion, let's recapitulate some important ideas that we've seen earlier.

In the *Introduction to Ada course* (page 115), we've seen that encapsulation plays an important role in modular programming. By using the private part of a package specification, we can disclose some information, but, at the same time, prevent that this information gets accessed where it shouldn't be used directly. Similarly, we've seen that we can use the private part of a package to distinguish between the *partial and full view* (page 312) of a data type.

The main application of private packages is to create private child packages, whose purpose is to serve as internal implementation packages within a package hierarchy. By doing so,

we can expose the internals to other public child packages, but prevent that external clients can directly access them.

As we'll see next, there are many rules that ensure that internal visibility is enforced for those private child packages. At the same time, the same rules ensure that private packages aren't visible outside of the package hierarchy.

Declaration and usage

We declare private packages by using the **private** keyword. For example, let's say we have a package named `Data_Processing`:

Listing 1198: `data_processing.ads`

```
1 package Data_Processing is
2
3   -- ...
4
5 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
 ↪Package_Decl
 MD5: 502811212890785d90c6f891d7f8e557

We simply write **private package** to declare a private child package named `Calculations`:

Listing 1199: `data_processing-calculations.ads`

```
1 private package Data_Processing.Calculations is
2
3   -- ...
4
5 end Data_Processing.Calculations;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
 ↪Package_Decl
 MD5: 20df8b2ac4c9aa93f03a12afd9b7ef30

Let's see a complete example:

Listing 1200: `data_processing.ads`

```
1 package Data_Processing is
2
3   type Data is private;
4
5   procedure Process (D : in out Data);
6
7 private
8
9   type Data is null record;
10
11 end Data_Processing;
```

Listing 1201: `data_processing-calculations.ads`

```
1 private package Data_Processing.Calculations is
2
```

(continues on next page)

(continued from previous page)

```
3   procedure Calculate (D : in out Data);
4
5 end Data_Processing.Calculations;
```

Listing 1202: data_processing.adb

```
1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package body Data_Processing is
5
6   procedure Process (D : in out Data) is
7   begin
8     Calculate (D);
9   end Process;
10
11 end Data_Processing;
```

Listing 1203: data_processing-calculations.adb

```
1 package body Data_Processing.Calculations is
2
3   procedure Calculate (D : in out Data) is
4   begin
5     -- Dummy implementation...
6     null;
7   end Calculate;
8
9 end Data_Processing.Calculations;
```

Listing 1204: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 procedure Test_Data_Processing is
4   D : Data;
5 begin
6   Process (D);
7 end Test_Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↳ Package
MD5: 3edd5f73938e809994347b5876014d0d

In this example, we refer to the private child package Calculations in the body of the Data_Processing package — by simply writing **with** Data_Processing.Calculations. After that, we can call the Calculate procedure normally in the Process procedure.

Private sibling packages

We can introduce another private package Advanced_Calculations as a child of Data_Processing and refer to the Calculations package in its specification:

Listing 1205: data_processing.ads

```
1 package Data_Processing is
2
```

(continues on next page)

(continued from previous page)

```

3  type Data is private;
4
5  procedure Process (D : in out Data);
6
7  private
8
9  type Data is null record;
10
11 end Data_Processing;

```

Listing 1206: data_processing-calculations.ads

```

1  private package Data_Processing.Calculations is
2
3  procedure Calculate (D : in out Data);
4
5  end Data_Processing.Calculations;

```

Listing 1207: data_processing-advanced_calculations.ads

```

1  with Data_Processing.Calculations;
2  use Data_Processing.Calculations;
3
4  private
5  package Data_Processing.Advanced_Calculations is
6
7  procedure Advanced_Calculate (D : in out Data)
8  renames Calculate;
9
10 end Data_Processing.Advanced_Calculations;

```

Listing 1208: data_processing.adb

```

1  with Data_Processing.Advanced_Calculations;
2  use Data_Processing.Advanced_Calculations;
3
4  package body Data_Processing is
5
6  procedure Process (D : in out Data) is
7  begin
8  Advanced_Calculate (D);
9  end Process;
10
11 end Data_Processing;

```

Listing 1209: data_processing-calculations.adb

```

1  package body Data_Processing.Calculations is
2
3  procedure Calculate (D : in out Data) is
4  begin
5  -- Dummy implementation...
6  null;
7  end Calculate;
8
9  end Data_Processing.Calculations;

```

Listing 1210: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 procedure Test_Data_Processing is
4   D : Data;
5 begin
6   Process (D);
7 end Test_Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↳Package_2
MD5: 32fc76ae13f1eecdd854a029793034d8

Note that, in the body of the `Data_Processing` package, we're now referring to the new `Advanced_Calculations` package instead of the `Calculations` package.

Referring to a private child package in the specification of another private child package is OK, but we cannot do the same in the specification of a *non-private* package. For example, let's change the specification of the `Advanced_Calculations` and make it *non-private*:

Listing 1211: data_processing-advanced_calculations.ads

```
1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package Data_Processing.Advanced_Calculations is
5
6   procedure Advanced_Calculate (D : in out Data)
7     renames Calculate;
8
9 end Data_Processing.Advanced_Calculations;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↳Package_2
MD5: 27fd3bdb063a1led7797cc44fa1e8349

Build output

```
data_processing-advanced_calculations.ads:1:06: error: current unit must also be_
↳private descendant of "Data_Processing"
gprbuild: *** compilation phase failed
```

Now, the compilation doesn't work anymore. However, we could still refer to `Calculations` packages in the body of the `Advanced_Calculations` package:

Listing 1212: data_processing-advanced_calculations.ads

```
1 package Data_Processing.Advanced_Calculations is
2
3   procedure Advanced_Calculate (D : in out Data);
4
5 end Data_Processing.Advanced_Calculations;
```

Listing 1213: data_processing-advanced_calculations.adb

```

1 with Data_Processing.Calculations;
2 use Data_Processing.Calculations;
3
4 package body Data_Processing.Advanced_Calculations
5 is
6
7     procedure Advanced_Calculate (D : in out Data)
8     is
9     begin
10         Calculate (D);
11     end Advanced_Calculate;
12
13 end Data_Processing.Advanced_Calculations;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
 ↳Package_2
 MD5: 3f37c129a6994c6b71a25ad17dcb440e

This works fine as expected: we can refer to private child packages in the body of another package — as long as both packages belong to the same package tree.

Outside the package tree

While we can use a with-clause of a private child package in the body of the Data_Processing package, we cannot do the same outside the package tree. For example, we cannot refer to it in the Test_Data_Processing procedure:

Listing 1214: test_data_processing.adb

```

1 with Data_Processing; use Data_Processing;
2
3 with Data_Processing.Calculations;
4 use Data_Processing.Calculations;
5
6 procedure Test_Data_Processing is
7     D : Data;
8 begin
9     Calculate (D);
10 end Test_Data_Processing;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
 ↳Package
 MD5: c844327995b28d60c9a79b138a0f21d2

Build output

```

test_data_processing.adb:3:06: error: unit in with clause is private child unit
test_data_processing.adb:3:06: error: current unit must also have parent "Data_
↳Processing"
gprbuild: *** compilation phase failed

```

As expected, we get a compilation error because Calculations is only accessible within the Data_Processing, but not in the Test_Data_Processing procedure.

The same restrictions apply to child packages of private packages. For example, if we

implement a child package of the Calculations package — let's name it Calculations.Child —, we cannot refer to it in the Test_Data_Processing procedure:

Listing 1215: data_processing-calculations-child.ads

```
1 package Data_Processing.Calculations.Child is
2
3     procedure Process (D : in out Data);
4
5 end Data_Processing.Calculations.Child;
```

Listing 1216: data_processing-calculations-child.adb

```
1 package body Data_Processing.Calculations.Child is
2
3     procedure Process (D : in out Data) is
4     begin
5         Calculate (D);
6     end Process;
7
8 end Data_Processing.Calculations.Child;
```

Listing 1217: test_data_processing.adb

```
1 with Data_Processing; use Data_Processing;
2
3 with Data_Processing.Calculations.Child;
4 use Data_Processing.Calculations.Child;
5
6 procedure Test_Data_Processing is
7     D : Data;
8 begin
9     Calculate (D);
10 end Test_Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
↳ Package
MD5: 2eaf23ddbab72578246ac07424008d9d

Build output

```
test_data_processing.adb:3:06: error: unit in with clause is private child unit
test_data_processing.adb:3:06: error: current unit must also have parent "Data_
↳ Processing"
test_data_processing.adb:9:04: error: "Calculate" is not visible
test_data_processing.adb:9:04: error: non-visible declaration at data_processing-
↳ calculations.ads:3
gprbuild: *** compilation phase failed
```

Again, as expected, we get an error because Calculations.Child — being a child of a private package — has the same restricted view as its parent package. Therefore, it cannot be visible in the Test_Data_Processing procedure as well. We'll discuss more about visibility [later](#) (page 860).

Note that subprograms can also be declared private. We'll see this [in another section](#) (page 875).

Important

We've discussed package renaming *in a previous section* (page 841). We can rename a package as a private package, too. For example:

Listing 1218: driver_m1.ads

```
1 package Driver_M1 is
2
3 end Driver_M1;
```

Listing 1219: drivers.ads

```
1 package Drivers
2   with Pure is
3
4 end Drivers;
```

Listing 1220: drivers-m1.ads

```
1 with Driver_M1;
2
3 private package Drivers.M1 renames Driver_M1;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
Package_Renaming

MD5: c03584dc26abb108c9c04074234b9637

Obviously, Drivers.M1 has the same restrictions as any private package:

Listing 1221: test_driver.adb

```
1 with Driver_M1;
2 with Drivers.M1;
3
4 procedure Test_Driver is
5 begin
6   null;
7 end Test_Driver;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_Packages.Private_
Package_Renaming

MD5: 55415978604ccea4eeaeb02df13cd2f4

Build output

```
test_driver.adb:2:06: error: unit in with clause is private child unit
test_driver.adb:2:06: error: current unit must also have parent "Drivers"
gprbuild: *** compilation phase failed
```

As expected, although we can have the Driver_M1 package in a with clause of the Test_Driver procedure, we cannot do the same in the case of the Drivers.M1 package because it is private.

In the Ada Reference Manual

- 10.1.1 Compilation Units - Library Units²³⁵

²³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

27.1.3 Private with clauses

Definition and usage

A private with clause allows us to refer to a package in the private part of another package. For example, if we want to refer to package P in the private part of Data, we can write **private with P**:

Listing 1222: p.ads

```
1 package P is
2
3     type T is null record;
4
5 end P;
```

Listing 1223: data.ads

```
1 private with P;
2
3 package Data is
4
5     type T2 is private;
6
7 private
8
9     -- Information from P is
10    -- visible here
11    type T2 is new P.T;
12
13 end Data;
```

Listing 1224: main.adb

```
1 with Data; use Data;
2
3 procedure Main is
4     A : T2;
5 begin
6     null;
7 end Main;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_
↳ Private_With_Clause
MD5: d0705add0dd7861c83822b0d35dacba4

As you can see in the example, as the information from P is available in the private part of Data, we can derive a new type T2 based on T from P. However, we cannot do the same in the visible part of Data:

Listing 1225: data.ads

```
1 private with P;
2
3 package Data is
4
5     -- ERROR: information from P
6     -- isn't visible here
7
8     type T2 is new P.T;
```

(continues on next page)

(continued from previous page)

```

9
10 end Data;
```

Code block metadata

```

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_
↳Private_With_Clause
MD5: b454e875f73432f5632a20ab40ae7da6
```

Build output

```

data.ads:8:19: error: "P" is not visible
data.ads:8:19: error: non-visible declaration at p.ads:1
gprbuild: *** compilation phase failed
```

Also, the information from P is available in the package body. For example, let's declare a Process procedure in the P package and use it in the body of the Data package:

Listing 1226: p.ads

```

1 package P is
2
3     type T is null record;
4
5     procedure Process (A : T) is null;
6
7 end P;
```

Listing 1227: data.ads

```

1 private with P;
2
3 package Data is
4
5     type T2 is private;
6
7     procedure Process (A : T2);
8
9 private
10
11     -- Information from P is
12     -- visible here
13     type T2 is new P.T;
14
15 end Data;
```

Listing 1228: data.adb

```

1 package body Data is
2
3     procedure Process (A : T2) is
4     begin
5         P.Process (P.T (A));
6     end Process;
7
8 end Data;
```


Listing 1229: main.adb

```
1 with Data; use Data;
2
3 procedure Main is
4     A : T2;
5 begin
6     null;
7 end Main;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Simple_
↳ Private_With_Clause
MD5: cecc09f95bd43dd7fd34a9e289bd2674
```

In the body of the Data, we can access information from the P package — as we do in the P.Process (P.T (A)) statement of the Process procedure.

Referring to private child package

There's one case where using a private with clause is the only way to refer to a package: when we want to refer to a private child package in another child package. For example, here we have a package P and its two child packages: **Private_Child** and **Public_Child**:

Listing 1230: p.ads

```
1 package P is
2
3 end P;
```

Listing 1231: p-private_child.ads

```
1 private package P.Private_Child is
2
3     type T is null record;
4
5 end P.Private_Child;
```

Listing 1232: p-public_child.ads

```
1 private with P.Private_Child;
2
3 package P.Public_Child is
4
5     type T2 is private;
6
7 private
8
9     type T2 is new P.Private_Child.T;
10
11 end P.Public_Child;
```

Listing 1233: test_parent_child.adb

```
1 with P.Public_Child; use P.Public_Child;
2
3 procedure Test_Parent_Child is
4     A : T2;
5 begin
```

(continues on next page)

(continued from previous page)

```

6   null;
7 end Test_Parent_Child;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Private_With_Clause
 MD5: a6028416a957184be55a54f96a319e61

In this example, we're referring to the P.**Private_Child** package in the P.Public_Child package. As expected, this works fine. However, using a *normal* with clause doesn't work in this case:

Listing 1234: p-public_child.ads

```

1 with P.Private_Child;
2
3 package P.Public_Child is
4
5     type T2 is private;
6
7 private
8
9     type T2 is new P.Private_Child.T;
10
11 end P.Public_Child;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Private_With_Clauses.Private_With_Clause
 MD5: 2f32f29ecb4ae13bb4487c94d3bf18d9

Build output

```

p-public_child.ads:1:06: error: current unit must also be private descendant of "P"
gprbuild: *** compilation phase failed

```

This gives an error because the information from the P.**Private_Child**, being a private child package, cannot be accessed in the public part of another child package. In summary, unless both packages are private packages, it's only possible to access the information from a private package in the private part of a non-private child package.

** In the Ada Reference Manual**

- 10.1.2 Context Clauses - With Clauses²³⁶

27.1.4 Limited Visibility

Sometimes, we might face the situation where two packages depend on information from each other. Let's consider a package A that depends on a package B, and vice-versa:

Listing 1235: a.ads

```

1 with B; use B;
2

```

(continues on next page)

²³⁶ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

(continued from previous page)

```
3 package A is
4
5     type T1 is record
6         Value : T2;
7     end record;
8
9 end A;
```

Listing 1236: b.ads

```
1 with A; use A;
2
3 package B is
4
5     type T2 is record
6         Value : T1;
7     end record;
8
9 end B;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Circular_
↳ Dependency
MD5: ae64f33706f1c58603aff2c33b02c910

Build output

```
b.ads:1:06: error: circular unit dependency
b.ads:1:06: error: "B (spec)" depends on "A (spec)"
b.ads:1:06: error: "A (spec)" depends on "B (spec)"
b.ads:1:06: error: "B (spec)" depends on "B (spec)"
gprbuild: *** compilation phase failed
```

Here, we have two *mutually dependent types* (page 448) T1 and T2, which are declared in two packages A and B that refer to each other. These with clauses constitute a circular dependency, so the compiler cannot compile either of those packages.

One way to solve this problem is by transforming this circular dependency into a partial dependency. We do this by limiting the visibility — using a limited with clause. To use a limited with clause for a package P, we simply write **limited with P**.

If a package A has limited visibility to a package B, then all types from package B are visible as if they had been declared as *incomplete types* (page 310). For the specific case of the previous source-code example, this would be the limited visibility to package B from package A's perspective:

```
package B is
    -- Incomplete type
    type T2;
end B;
```

As we've seen previously,

- we cannot declare objects of incomplete types, but we can declare access types and anonymous access objects of incomplete types. Also,
- we can use anonymous access types to declare *mutually dependent types* (page 448).

Keeping this information in mind, we can now correct the previous code by using limited with

clauses for package A and declaring the component of the T1 record using an anonymous access type:

Listing 1237: a.ads

```

1 limited with B;
2
3 package A is
4
5     type T1 is record
6         Ref : access B.T2;
7     end record;
8
9 end A;
```

Listing 1238: b.ads

```

1 with A; use A;
2
3 package B is
4
5     type T2 is record
6         Value : T1;
7     end record;
8
9 end B;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_Visibility
 MD5: 48591850665085a6fbb184f51b658a1b

As expected, we can now compile the code without issues.

Note that we can also use limited with clauses for both packages. If we do that, we must declare all components using anonymous access types:

Listing 1239: a.ads

```

1 limited with B;
2
3 package A is
4
5     type T1 is record
6         Ref : access B.T2;
7     end record;
8
9 end A;
```

Listing 1240: b.ads

```

1 limited with A;
2
3 package B is
4
5     type T2 is record
6         Ref : access A.T1;
7     end record;
8
9 end B;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↳Visibility_2
MD5: 3884086e89400245346acfbfbf0691906
```

Now, both packages A and B have limited visibility to each other.

In the Ada Reference Manual

- 10.1.2 Context Clauses - With Clauses²³⁷

Limited visibility and private with clauses

We can limit the visibility and use *private with clauses* (page 852) at the same time. For a package P, we do this by simply writing **limited private with** P.

Let's reuse the previous source-code example and convert types T1 and T2 to private types:

Listing 1241: a.ads

```
1  limited private with B;
2
3  package A is
4
5      type T1 is private;
6
7  private
8
9      -- Here, we have limited visibility
10     -- of package B
11
12     type T1 is record
13         Ref : access B.T2;
14     end record;
15
16 end A;
```

Listing 1242: b.ads

```
1  private with A;
2
3  package B is
4
5      type T2 is private;
6
7  private
8
9      use A;
10
11     -- Here, we have full visibility
12     -- of package A
13
14     type T2 is record
15         Value : T1;
16     end record;
17
18 end B;
```

Code block metadata

²³⁷ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↳Private_Visibility
MD5: b3ac546e2f55fb91229e834ca7a9783d
```

In this updated version of the source-code example, we have not only limited visibility to package B, but also, each package is just visible in the private part of the other package.

Limited visibility and other elements

It's important to mention that the limited visibility we've been discussing so far is restricted to type declarations — which are seen as incomplete types. In fact, when we use a limited with clause, all other declarations have no visibility at all! For example, let's say we have a package Info that declares a constant Zero_Const and a function Zero_Func:

Listing 1243: info.ads

```
1 package Info is
2
3     function Zero_Func return Integer is (0);
4
5     Zero_Const : constant := 0;
6
7 end Info;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↳Private_Visibility_Other_Elements
MD5: e9b01b4d59db5982532634f9162518ce
```

Also, let's say we want to use the information (from package Info) in package A. If we have limited visibility to package Info, however, this information won't be visible. For example:

Listing 1244: a.ads

```
1 limited private with Info;
2
3 package A is
4
5     type T1 is private;
6
7 private
8
9     type T1 is record
10         V : Integer := Info.Zero_Const;
11         W : Integer := Info.Zero_Func;
12     end record;
13
14 end A;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Modular_Prog.Packages.Limited_Visibility.Limited_
↳Private_Visibility_Other_Elements
MD5: 61ecb5dc2617eecac62a05d7d2c6c0df
```

Build output

```
a.ads:10:26: error: "Zero_Const" not declared in "Info"
a.ads:11:26: error: "Zero_Func" not declared in "Info"
gprbuild: *** compilation phase failed
```

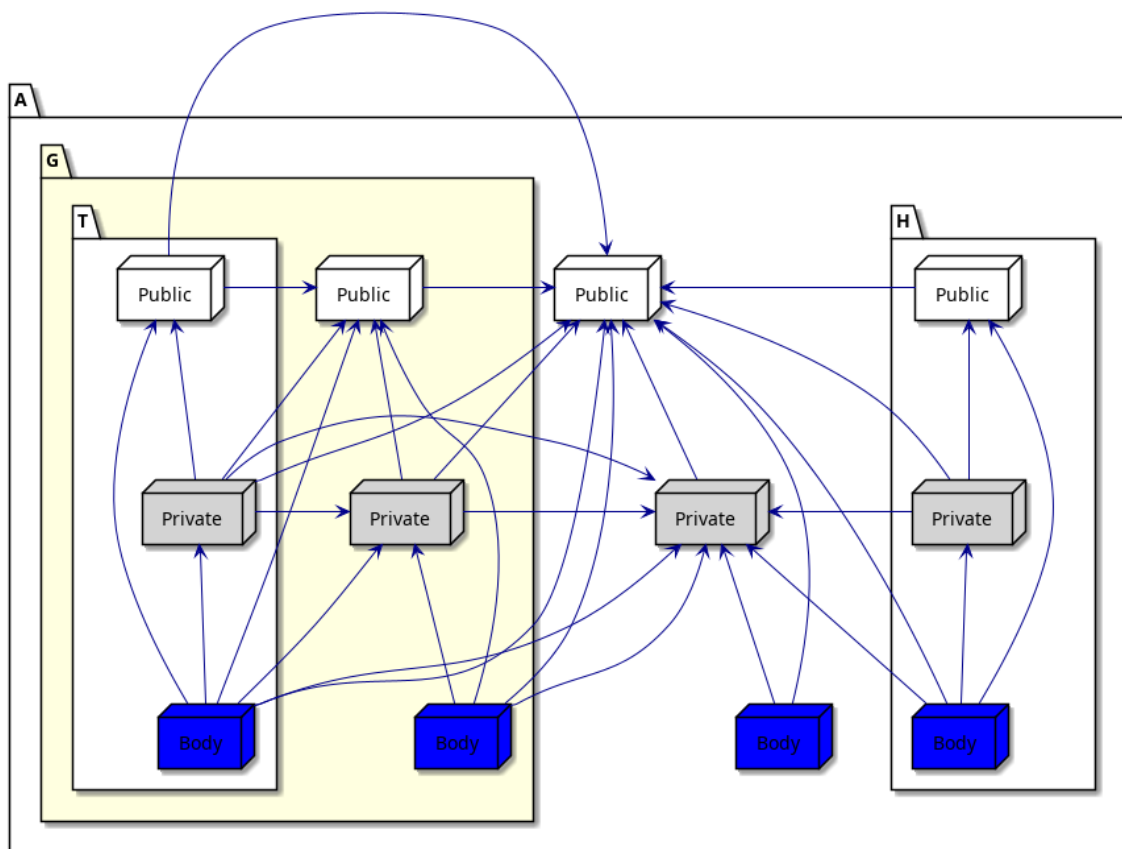
As expected, compilation fails because of the limited visibility — as `Zero_Const` and `Zero_Func` from the `Info` package are not visible in the private part of `A`. (Of course, if we revert to full visibility by simply removing the **limited** keyword from the example, the code compiles just fine.)

27.1.5 Visibility

In the previous sections, we already discussed visibility from various angles. However, it can be interesting to recapitulate this information with the help of diagrams that illustrate the different parts of a package and its relation with other units.

Automatic visibility

First, let's consider we have a package `A`, its children (`A.G` and `A.H`), and the grandchild `A.G.T`. As we've seen before, information of a parent package is automatically visible in its children. The following diagrams illustrates this:

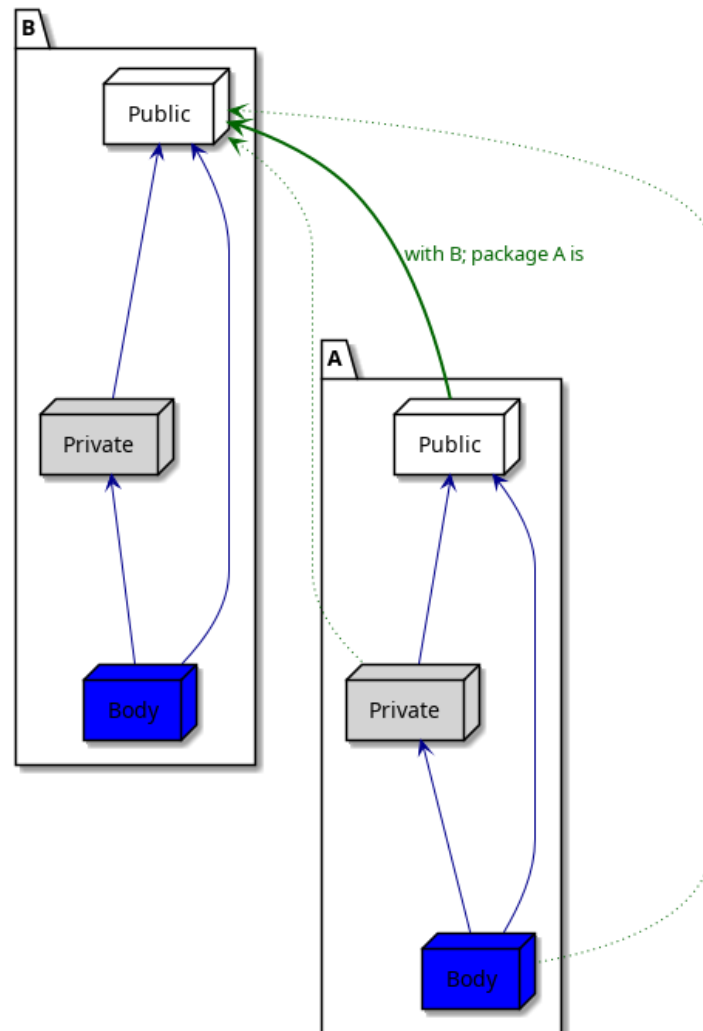


Because of this automatic visibility, many `with` clauses would be redundant in child packages. For example, we don't have to write `with A; package A.G is`, since the specification of package `A` is already visible in its child packages.

If we focus on package `A.G` (highlighted in the figure above), we see that it only has automatic visibility to its parent `A`, but not its child `A.G.T`. Also, it doesn't have visibility to its sibling `A.H`.

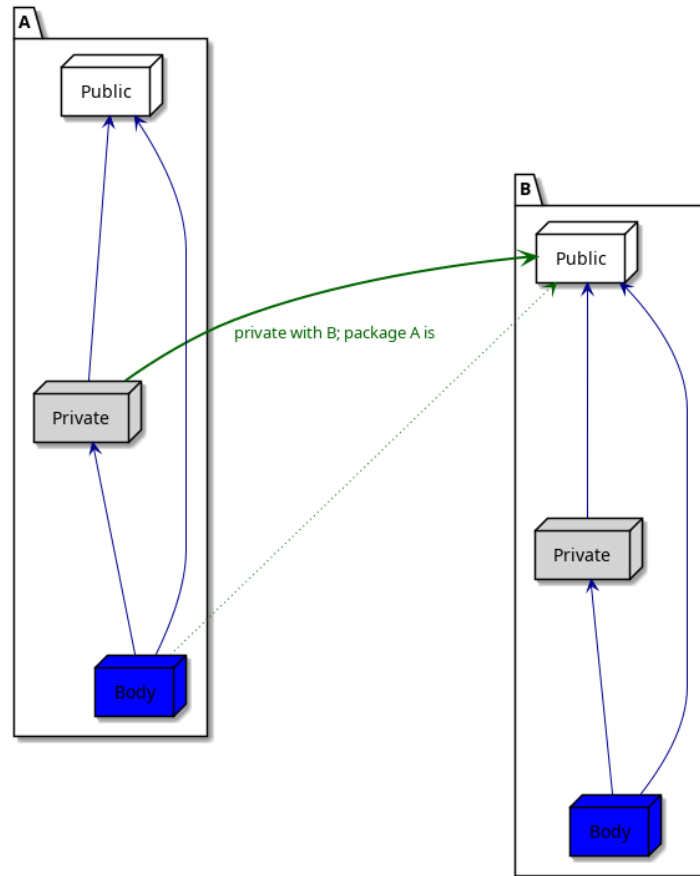
With clauses and visibility

In the rest of this section, we discuss all the situations where using with clauses is necessary to access the information of a package. Let's consider this example where we refer to a package B in the specification of a package A (using **with B**):

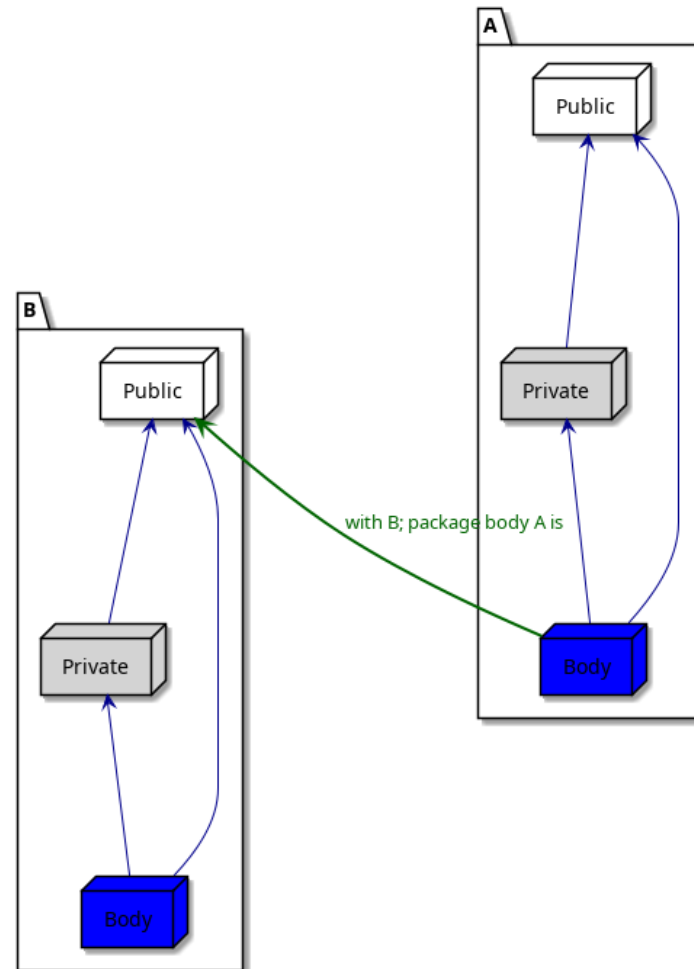


As we already know, the information from the public part of package B is visible in the public part of package A. In addition to that, it's also visible in the private part and in the body of package A. This is indicated by the dotted green arrows in the figure above.

Now, let's see the case where we refer to package B in the private part of package A (using **private with B**):



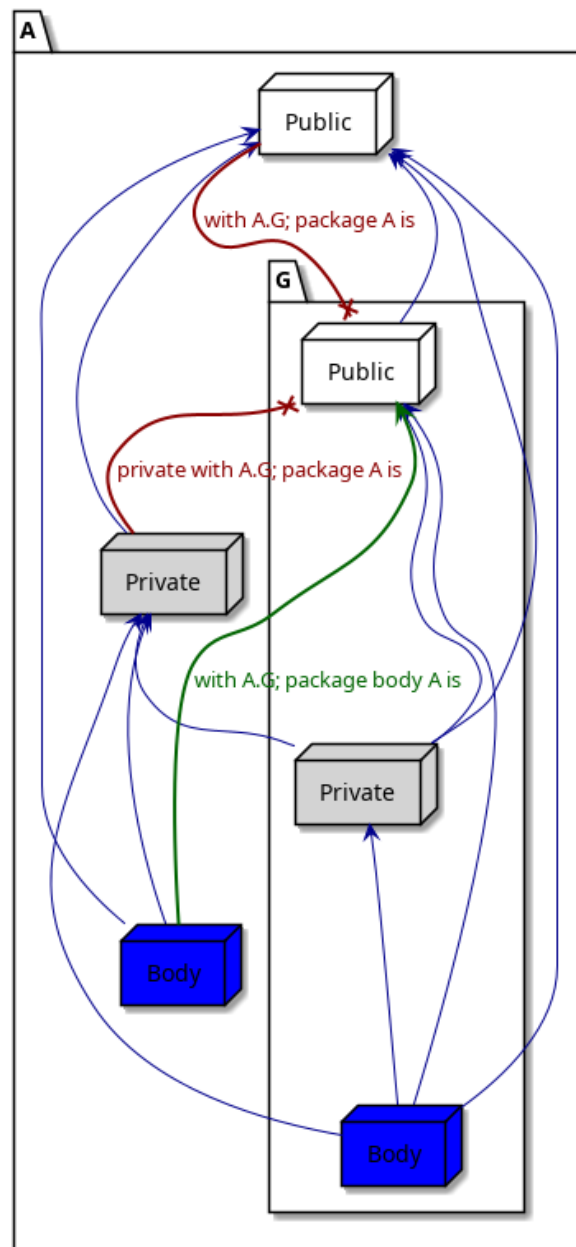
Here, the information is visible in the private part of package A, as well as in its body. Finally, let's see the case where we refer to package B in the body of package A:



Here, the information is only visible in the body of package A.

Circular dependency

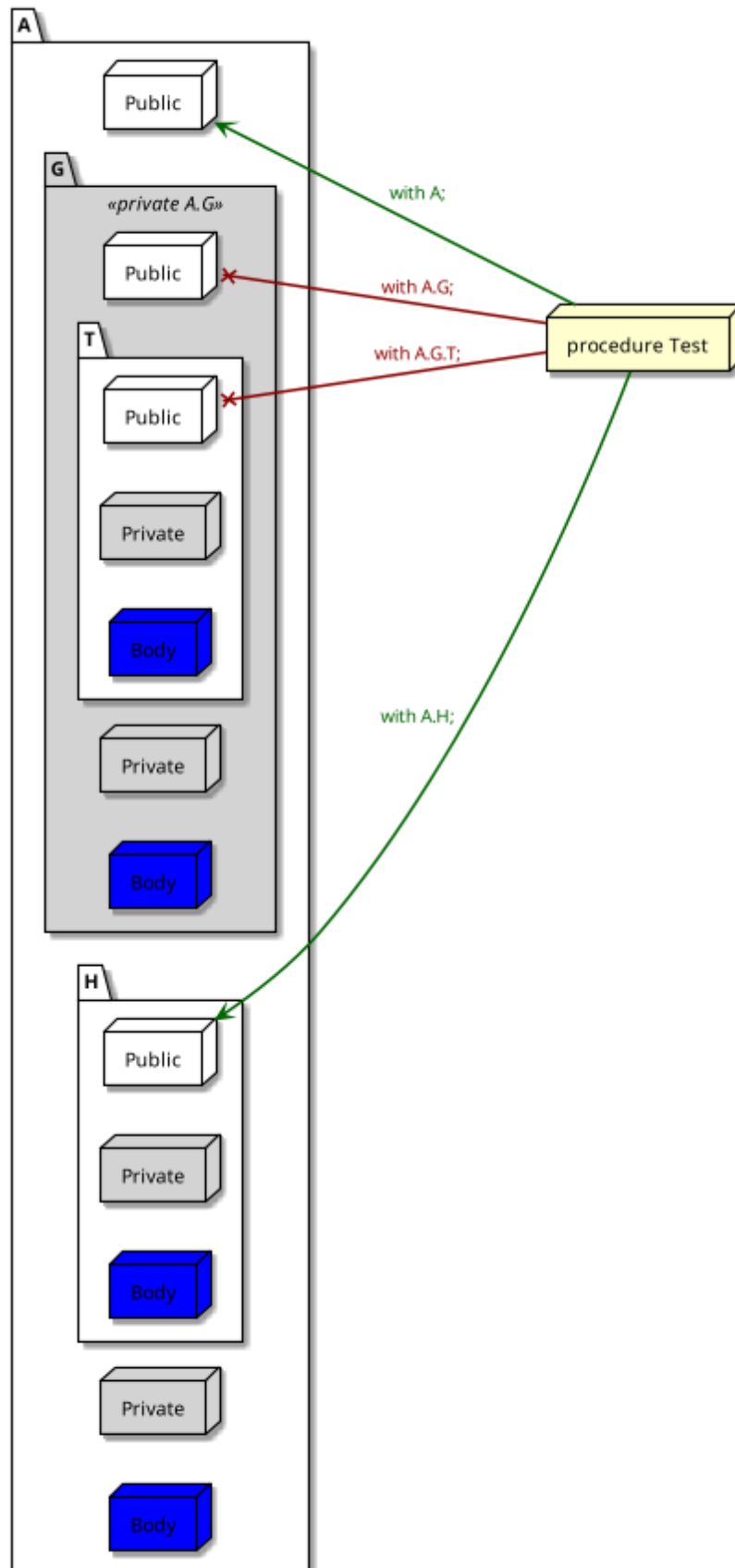
Let's return to package A and its descendants. As we've seen in previous sections, we cannot refer to a child package in the specification of its parent package because that would constitute circular dependency. (For example, we cannot write **with A.G;** **package A is.**) This situation — which causes a compilation error — is indicated by the red arrows in the figure below:



Note that referring to the child package A.G in the body of its parent is perfectly fine.

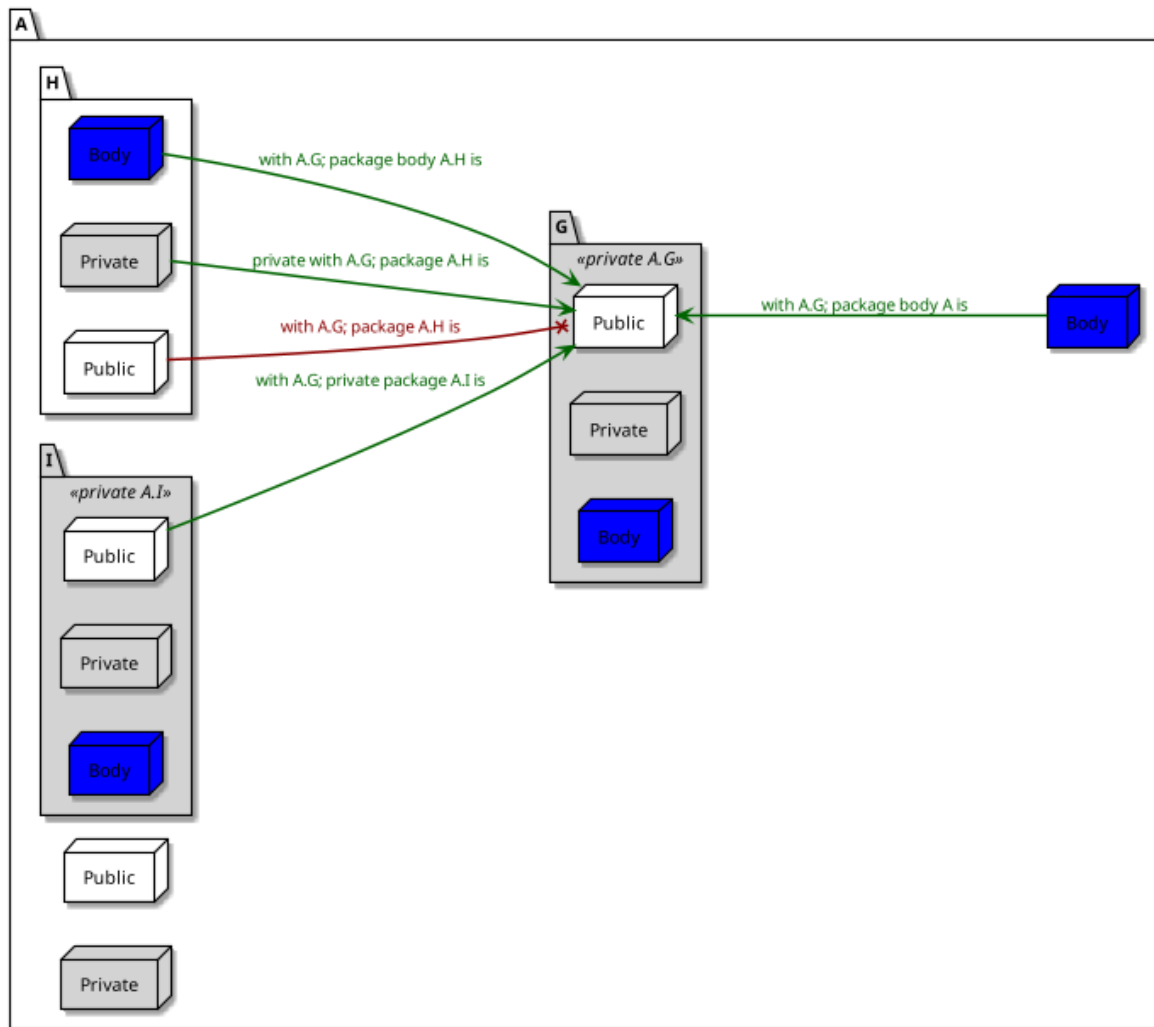
Private packages

The previous examples of this section only showed public packages. As we've seen before, we cannot refer to private packages outside of a package hierarchy, as we can see in the following example where we try to refer to package A and its descendants in the Test procedure:



As indicated by the red arrows, we cannot refer to the private child packages of A in the Test procedure, only the public child packages. Within the package hierarchy itself, we

cannot refer to the private package A.G in public sibling packages. For example:



Here, we cannot refer to the private package A.G in the public package A.H — as indicated by the red arrow. However, we can refer to the private package A.G in other private packages, such as A.I — as indicated by the green arrows.

27.1.6 Use type clause

Back in the *Introduction to Ada course* (page 37), we saw that use clauses provide direct visibility — in the scope where they're used — to the content of a package's visible part.

For example, consider this simple procedure:

Listing 1245: display_message.adb

```

1 with Ada.Text_IO;
2
3 procedure Display_Message is
4 begin
5   Ada.Text_IO.Put_Line ("Hello World!");
6 end Display_Message;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.No_Use_Clause
 MD5: 4c6ff19809c13ebd2fdafa482914e5f8

Runtime output

Hello World!

By adding `use Ada.Text_IO` to this code, we make the visible part of the `Ada.Text_IO` package directly visible in the scope of the `Display_Message` procedure, so we can now just write `Put_Line` instead of `Ada.Text_IO.Put_Line`:

Listing 1246: display_message.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Message is
4 begin
5   Put_Line ("Hello World!");
6 end Display_Message;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Clause
 MD5: b105a777a1afd79008f8580cda432cfe

Runtime output

Hello World!

In this section, we discuss another example of use clauses. In addition, we introduce two specific forms of use clauses: `use type` and `use all type`.

In the Ada Reference Manual

- [8.4 Use Clauses](#)²³⁸

Another use clause example

Let's now consider a simple package called `Points`, which contains the declaration of the `Point` type and two primitive: an `Init` function and an addition operator.

Listing 1247: points.ads

```

1 package Points is
2
3   type Point is private;
4
5   function Init return Point;
6
7   function "+" (P : Point;
8               I : Integer) return Point;
9
10 private
11
12   type Point is record
13     X, Y : Integer;
14   end record;
```

(continues on next page)

²³⁸ <http://www.ada-auth.org/standards/22rm/html/RM-8-4.html>

(continued from previous page)

```
16  function Init return Point is (0, 0);
17
18  function "+" (P : Point;
19              I : Integer) return Point is
20      (P.X + I, P.Y + I);
21
22  end Points;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: 1a43740d7231a3cc497e778866a12c55

We can implement a simple procedure that makes use of this package:

Listing 1248: show_point.adb

```
1  with Points; use Points;
2
3  procedure Show_Point is
4      P : Point;
5  begin
6      P := Init;
7      P := P + 1;
8  end Show_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: f5d44dd1fee8cf4d1a7e730f9a7c64cc

Here, we have a use clause, so we have direct visibility to the content of Points's visible part.

Visibility and Readability

In certain situations, however, we might want to avoid the use clause. If that's the case, we can rewrite the previous implementation by removing the use clause and specifying the Points package in the prefixed form:

Listing 1249: show_point.adb

```
1  with Points;
2
3  procedure Show_Point is
4      P : Points.Point;
5  begin
6      P := Points.Init;
7      P := Points."+" (P, 1);
8  end Show_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: ca896b456a90c19b29ec4f262144c131

Although this code is correct, it might be difficult to read, as we have to specify the package whenever we're referring to a type or a subprogram from that package. Even worse: we now have to write operators in the prefixed form — such as Points."+" (P, 1).

use type

As a compromise, we can have direct visibility to the operators of a certain type. We do this by using a use clause in the form **use type**. This allows us to simplify the previous example:

Listing 1250: show_point.adb

```

1 with Points;
2
3 procedure Show_Point is
4     use type Points.Point;
5
6     P : Points.Point;
7 begin
8     P := Points.Init;
9     P := P + 1;
10 end Show_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: a9527276c27a67be8b5a59efcf6e5cfd

Note that **use type** just gives us direct visibility to the operators of a certain type, but not other primitives. For this reason, we still have to write `Points.Init` in the code example.

use all type

If we want to have direct visibility to all primitives of a certain type (and not just its operators), we need to write a use clause in the form **use all type**. This allows us to simplify the previous example even further:

Listing 1251: show_point.adb

```

1 with Points;
2
3 procedure Show_Point is
4     use all type Points.Point;
5
6     P : Points.Point;
7 begin
8     P := Init;
9     P := P + 1;
10 end Show_Point;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Type_Clause.Use_Type_Clause
MD5: 4a8f6edd4e1811c4e8acb24393690282

Now, we've removed the prefix from all operations on the `P` variable.

27.1.7 Use clauses and naming conflicts

Visibility issues may arise when we have multiple use clauses. For instance, we might have types with the same name declared in multiple packages. This constitutes a naming conflict; in this case, the types become hidden — so they're not directly visible anymore, even if we have a use clause.

In the Ada Reference Manual

- 8.4 Use Clauses²³⁹

Code example

Let's start with a code example. First, we declare and implement a generic procedure that shows the value of a Complex object:

Listing 1252: show_any_complex.ads

```
1 with Ada.Numerics.Generic_Complex_Types;
2
3 generic
4   with package Complex_Types is new
5     Ada.Numerics.Generic_Complex_Types (<>);
6 procedure Show_Any_Complex
7   (Msg : String;
8    Val : Complex_Types.Complex);
```

Listing 1253: show_any_complex.adb

```
1 with Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3
4 procedure Show_Any_Complex
5   (Msg : String;
6    Val : Complex_Types.Complex)
7 is
8   package Complex_Float_Types_IO is new
9     Ada.Text_IO.Complex_IO (Complex_Types);
10  use Complex_Float_Types_IO;
11
12  use Ada.Text_IO;
13 begin
14   Put (Msg & " ");
15   Put (Val);
16   New_Line;
17 end Show_Any_Complex;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.

↪ Use_Type_Clause_Complex_Types

MD5: 2527291906d3a600eecd6d36e4359c1a

Then, we implement a test procedure where we declare the Complex_Float_Types package as an instance of the **Generic_Complex_Types** package:

Listing 1254: show_use.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2
3 with Ada.Numerics.Generic_Complex_Types;
4
5 with Show_Any_Complex;
6
7 procedure Show_Use is
```

(continues on next page)

²³⁹ <http://www.ada-auth.org/standards/22rm/html/RM-8-4.html>

(continued from previous page)

```

8  package Complex_Float_Types is new
9      Ada.Numerics.Generic_Complex_Types
10     (Real => Float);
11  use Complex_Float_Types;
12
13  procedure Show_Complex_Float is new
14      Show_Any_Complex (Complex_Float_Types);
15
16  C, D, X : Complex;
17  begin
18      C := Compose_From_Polar (3.0, Pi / 2.0);
19      D := Compose_From_Polar (5.0, Pi / 2.0);
20      X := C + D;
21
22      Show_Complex_Float ("C:", C);
23      Show_Complex_Float ("D:", D);
24      Show_Complex_Float ("X:", X);
25  end Show_Use;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
 Use_Type_Clause_Complex_Types
 MD5: cc2a612c9884539f33154680854a4c82

Runtime output

```

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)

```

In this example, we declare variables of the Complex type, initialize them and use them in operations. Note that we have direct visibility to the package instance because we've added a simple use clause after the package instantiation — see **use** Complex_Float_Types in the example.

Naming conflict

Now, let's add the declaration of the Complex_Long_Float_Types package — a second instantiation of the **Generic_Complex_Types** package — to the code example:

Listing 1255: show_use.adb

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;
6
7  procedure Show_Use is
8      package Complex_Float_Types is new
9          Ada.Numerics.Generic_Complex_Types
10         (Real => Float);
11  use Complex_Float_Types;
12
13  package Complex_Long_Float_Types is new
14      Ada.Numerics.Generic_Complex_Types
15      (Real => Long_Float);
16  use Complex_Long_Float_Types;
17

```

(continues on next page)

(continued from previous page)

```

18  procedure Show_Complex_Float is new
19      Show_Any_Complex (Complex_Float_Types);
20
21  C, D, X : Complex;
22  --      ^ ERROR: type is hidden!
23  begin
24      C := Compose_From_Polar (3.0, Pi / 2.0);
25      D := Compose_From_Polar (5.0, Pi / 2.0);
26      X := C + D;
27
28      Show_Complex_Float ("C:", C);
29      Show_Complex_Float ("D:", D);
30      Show_Complex_Float ("X:", X);
31  end Show_Use;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪ Use_Type_Clause_Complex_Types
MD5: 30b562e2f81ae62912ec4e067150d5cd

```

Build output

```

show_use.adb:21:14: error: "Complex" is not visible
show_use.adb:21:14: error: multiple use clauses cause hiding
show_use.adb:21:14: error: hidden declaration at a-ngcoty.ads:42, instance at line ↪
↪ 13
show_use.adb:21:14: error: hidden declaration at a-ngcoty.ads:42, instance at line ↪
↪ 8
gprbuild: *** compilation phase failed

```

This example doesn't compile because we have direct visibility to both `Complex_Float_Types` and `Complex_Long_Float_Types` packages, and both of them declare the `Complex` type. In this case, the type declaration becomes hidden, as the compiler cannot decide which declaration of `Complex` it should take.

Circumventing naming conflicts

As we know, a simple fix for this compilation error is to add the package prefix in the variable declaration:

Listing 1256: `show_use.adb`

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;
6
7  procedure Show_Use is
8      package Complex_Float_Types is new
9          Ada.Numerics.Generic_Complex_Types
10             (Real => Float);
11      use Complex_Float_Types;
12
13      package Complex_Long_Float_Types is new
14          Ada.Numerics.Generic_Complex_Types
15             (Real => Long_Float);
16      use Complex_Long_Float_Types;
17

```

(continues on next page)

(continued from previous page)

```

18  procedure Show_Complex_Float is new
19      Show_Any_Complex (Complex_Float_Types);
20
21  C, D, X : Complex_Float_Types.Complex;
22  --      ^ SOLVED: package is now specified.
23  begin
24      C := Compose_From_Polar (3.0, Pi / 2.0);
25      D := Compose_From_Polar (5.0, Pi / 2.0);
26      X := C + D;
27
28      Show_Complex_Float ("C:", C);
29      Show_Complex_Float ("D:", D);
30      Show_Complex_Float ("X:", X);
31  end Show_Use;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
 ↳ Use_Type_Clause_Complex_Types
 MD5: 0b3285364ea0188a678db2fc406741b8

Runtime output

```

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)

```

Another possibility is to write a use clause in the form **use all type**:

Listing 1257: show_use.adb

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  with Ada.Numerics.Generic_Complex_Types;
4
5  with Show_Any_Complex;
6
7  procedure Show_Use is
8      package Complex_Float_Types is new
9          Ada.Numerics.Generic_Complex_Types
10             (Real => Float);
11      use all type Complex_Float_Types.Complex;
12
13      package Complex_Long_Float_Types is new
14          Ada.Numerics.Generic_Complex_Types
15             (Real => Long_Float);
16      use all type Complex_Long_Float_Types.Complex;
17
18      procedure Show_Complex_Float is new
19          Show_Any_Complex (Complex_Float_Types);
20
21      C, D, X : Complex_Float_Types.Complex;
22  begin
23      C := Compose_From_Polar (3.0, Pi / 2.0);
24      D := Compose_From_Polar (5.0, Pi / 2.0);
25      X := C + D;
26
27      Show_Complex_Float ("C:", C);
28      Show_Complex_Float ("D:", D);
29      Show_Complex_Float ("X:", X);
30  end Show_Use;

```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪ Use_Type_Clause_Complex_Types
MD5: 90333ff41e25afb1399f7f94f7e2b566

Runtime output

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)

For the sake of completeness, let's declare and use variables of both Complex types:

Listing 1258: show_use.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2
3 with Ada.Numerics.Generic_Complex_Types;
4
5 with Show_Any_Complex;
6
7 procedure Show_Use is
8   package Complex_Float_Types is new
9     Ada.Numerics.Generic_Complex_Types
10      (Real => Float);
11   use all type Complex_Float_Types.Complex;
12
13   package Complex_Long_Float_Types is new
14     Ada.Numerics.Generic_Complex_Types
15      (Real => Long_Float);
16   use all type Complex_Long_Float_Types.Complex;
17
18   procedure Show_Complex_Float is new
19     Show_Any_Complex (Complex_Float_Types);
20
21   procedure Show_Complex_Long_Float is new
22     Show_Any_Complex (Complex_Long_Float_Types);
23
24   C, D, X : Complex_Float_Types.Complex;
25   E, F, Y : Complex_Long_Float_Types.Complex;
26 begin
27   C := Compose_From_Polar (3.0, Pi / 2.0);
28   D := Compose_From_Polar (5.0, Pi / 2.0);
29   X := C + D;
30
31   Show_Complex_Float ("C:", C);
32   Show_Complex_Float ("D:", D);
33   Show_Complex_Float ("X:", X);
34
35   E := Compose_From_Polar (3.0, Pi / 2.0);
36   F := Compose_From_Polar (5.0, Pi / 2.0);
37   Y := E + F;
38
39   Show_Complex_Long_Float ("E:", E);
40   Show_Complex_Long_Float ("F:", F);
41   Show_Complex_Long_Float ("Y:", Y);
42 end Show_Use;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Packages.Use_Clause_Naming_Conflicts.
↪ Use_Type_Clause_Complex_Types

(continues on next page)

(continued from previous page)

MD5: 48f31250116f107d3143703debb3107d

Runtime output

```

C: (-1.31134E-07, 3.00000E+00)
D: (-2.18557E-07, 5.00000E+00)
X: (-3.49691E-07, 8.00000E+00)
E: ( 1.83697019872103E-16, 3.00000000000000E+00)
F: ( 3.06161699786838E-16, 5.00000000000000E+00)
Y: ( 4.89858719658941E-16, 8.00000000000000E+00)

```

As expected, the code compiles correctly.

27.2 Subprograms and Modularity

27.2.1 Private subprograms

We've seen *previously* (page 844) that we can declare private packages. Because packages and subprograms can both be library units, we can declare private subprograms as well. We do this by using the **private** keyword. For example:

Listing 1259: test.ads

```
1 private procedure Test;
```

Listing 1260: test.adb

```

1 procedure Test is
2 begin
3     null;
4 end Test;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↳Subprograms.Private_Test_Procedure
MD5: 2ea1770a5fd5dee40f015b9d33d2f309

```

Such a subprogram as the one above isn't really useful. For example, we cannot write a with clause that refers to the Test procedure, as it's not visible anywhere:

Listing 1261: show_test.adb

```

1 with Test;
2
3 procedure Show_Test is
4 begin
5     Test;
6 end Show_Test;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↳Subprograms.Private_Test_Procedure
MD5: 0702378a034f65a69a4c5b5258f7b32e

```

Build output

```
show_test.adb:1:06: error: current unit must also be private descendant of
↳ "Standard"
gprbuild: *** compilation phase failed
```

As expected, since `Test` is private, we get a compilation error because this procedure cannot be referenced in the `Show_Test` procedure.

In the Ada Reference Manual

- 10.1.1 Compilation Units - Library Units²⁴⁰
- 10.1.2 Context Clauses - With Clauses²⁴¹

Private subprograms of a package

A more useful example is to declare private subprograms of a package. For example:

Listing 1262: `data_processing.ads`

```
1 package Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7 private
8
9     type Data is record
10         F : Float;
11     end record;
12
13 end Data_Processing;
```

Listing 1263: `data_processing.adb`

```
1 with Data_Processing.Calculate;
2
3 package body Data_Processing is
4
5     procedure Process (D : in out Data) is
6     begin
7         Calculate (D);
8     end Process;
9
10 end Data_Processing;
```

Listing 1264: `data_processing-calculate.ads`

```
1 private
2 procedure Data_Processing.Calculate
3     (D : in out Data);
```

Listing 1265: `data_processing-calculate.adb`

```
1 procedure Data_Processing.Calculate
2     (D : in out Data)
```

(continues on next page)

²⁴⁰ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-1.html>

²⁴¹ <http://www.ada-auth.org/standards/22rm/html/RM-10-1-2.html>

(continued from previous page)

```

3 is
4 begin
5     -- Dummy implementation...
6     D.F := 0.0;
7 end Data_Processing.Calculate;

```

Listing 1266: test_data_processing.adb

```

1 with Data_Processing; use Data_Processing;
2
3 procedure Test_Data_Processing is
4     D : Data;
5 begin
6     Process (D);
7 end Test_Data_Processing;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↳Subprograms.Private_Package_Procedure
MD5: 0f6af1b02f37e011abac5b2a6dfc482d

```

In this example, we declare `Calculate` as a private procedure of the `Data_Processing` package. Therefore, it's visible in that package (but not in the `Test_Data_Processing` procedure). Also, in the `Calculate` procedure, we're able to initialize the private component `F` of the `D` object because the child subprogram has access to the private part of its parent package.

Private subprograms and private packages

We can also use private subprograms to test private packages. As we know, in most cases, we cannot access private packages in external clients — such as external subprograms. However, by declaring a subprogram private, we're allowed to access private packages. This can be very useful to create applications that we can use to test private packages. (Note that these applications must be library-level parameterless subprograms, because only those can be main programs.)

Let's see an example:

Listing 1267: private_data_processing.ads

```

1 private package Private_Data_Processing is
2
3     type Data is private;
4
5     procedure Process (D : in out Data);
6
7 private
8
9     type Data is record
10         F : Float;
11     end record;
12
13 end Private_Data_Processing;

```

Listing 1268: private_data_processing.adb

```

1 package body Private_Data_Processing is
2

```

(continues on next page)

(continued from previous page)

```
3  procedure Process (D : in out Data) is
4  begin
5      D.F := 0.0;
6  end Process;
7
8  end Private_Data_Processing;
```

Listing 1269: test_private_data_processing.ads

```
1  private procedure Test_Private_Data_Processing;
```

Listing 1270: test_private_data_processing.adb

```
1  with Private_Data_Processing;
2  use Private_Data_Processing;
3
4  procedure Test_Private_Data_Processing is
5      D : Data;
6  begin
7      Process (D);
8  end Test_Private_Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
↳ Subprograms.Private_Subprogram_Private_Package
MD5: 3527e54f99eb2cb52317c987b499caaf

In this code example, we have the private **Private_Data_Processing** package. In order to test it, we implement the private procedure **Test_Private_Data_Processing**. The fact that this procedure is private allows us to use the **Private_Data_Processing** package as if it was a non-private package. We then use the private **Test_Private_Data_Processing** procedure as our main application, so we can run it to test application the private package.

Child subprograms of private packages

We could also implement the **Test** subprogram that we use to test a private package **P** as a child subprogram of that package. In other words, we could write a procedure **P.Test** and use it as our main application. The advantage here is that this allows us to access the private part of the parent package **P** in the test procedure.

Let's rewrite the **Test_Private_Data_Processing** procedure from the previous example as the child procedure **Private_Data_Processing.Test**:

Listing 1271: private_data_processing.ads

```
1  private package Private_Data_Processing is
2
3      type Data is private;
4
5      procedure Process (D : in out Data);
6
7  private
8
9      type Data is record
10         F : Float;
11     end record;
12
13  end Private_Data_Processing;
```

Listing 1272: private_data_processing.adb

```

1 package body Private_Data_Processing is
2
3     procedure Process (D : in out Data) is
4     begin
5         null;
6     end Process;
7
8 end Private_Data_Processing;
```

Listing 1273: private_data_processing-test.ads

```

1 procedure Private_Data_Processing.Test;
```

Listing 1274: private_data_processing-test.adb

```

1 procedure Private_Data_Processing.Test is
2     D : Data := (F => 0.0);
3 begin
4     Process (D);
5 end Private_Data_Processing.Test;
```

Code block metadata

Project: Courses.Advanced_Ada.Modular_Prog.Subprograms_Modularity.Private_
 ↪Subprograms.Private_Package_Child_Subprogram
 MD5: 0726f5890a5b3847244d1ae08989e158

In this code example, we now implement the Test procedure as a child of the **Private_Data_Processing** package. In this procedure, we're able to initialize the private component F of the D object. As we know, this initialization of a private component wouldn't be possible if Test wasn't a child procedure. (For instance, writing such an initialization in the Test_Private_Data_Processing procedure from the previous code example would trigger a compilation error.)

RESOURCE MANAGEMENT

28.1 Access Types

We discussed access types back in the *Introduction to Ada course* (page 95). In this chapter, we discuss further details about access types and techniques when using them. Before we dig into details, however, we're going to make sure we understand the terminology.

28.1.1 Access types: Terminology

In this section, we discuss some of the terminology associated with access types. Usually, the terms used in Ada when discussing references and dynamic memory allocation are different than the ones you might encounter in other languages, so it's necessary you understand what each term means.

Access type, designated subtype and profile

The first term we encounter is (obviously) *access type*, which is a type that provides us access to an object or a subprogram. We declare access types by using the **access** keyword:

Listing 1275: show_access_type_declaration.ads

```
1 package Show_Access_Type_Declaration is
2
3     --
4     --   Declaring access types:
5     --
6
7     --   Access-to-object type
8     type Integer_Access is access Integer;
9
10    --   Access-to-subprogram type
11    type Init_Integer_Access is access
12        function return Integer;
13
14 end Show_Access_Type_Declaration;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Access_
Type_Declaration
MD5: 64e4e0847a73a9ed23e29e09798934de

Here, we're declaring two access types: the access-to-object type `Integer_Access` and the access-to-subprogram type `Init_Integer_Access`. (We discuss access-to-subprogram types *later on* (page 964)).

In the declaration of an access type, we always specify — after the **access** keyword — the kind of thing we want to designate. In the case of an access-to-object type declaration,

we declare a subtype we want to access, which is known as the *designated subtype* of an access type. In the case of an access-to-subprogram type declaration, the subprogram prototype is known as the *designated profile*.

In our previous code example, **Integer** is the designated subtype of the `Integer_Access` type, and **function return Integer** is the designated profile of the `Init_Integer_Access` type.

Important

In contrast to other programming languages, an access type is not a pointer, and it doesn't just indicate an address in memory. We discuss more about [addresses](#) (page 993) later on.

Access object and designated object

We use an access-to-object type by first declaring a variable (or constant) of an access type and then allocating an object. (This is actually just one way of using access types; we discuss other methods later in this chapter.) The actual variable or constant of an access type is called *access object*, while the object we allocate (via **new**) is the *designated object*.

For example:

Listing 1276: `show_simple_allocation.adb`

```
1 procedure Show_Simple_Allocation is
2
3   -- Access-to-object type
4   type Integer_Access is access Integer;
5
6   -- Access object
7   I1 : Integer_Access;
8
9 begin
10  I1 := new Integer;
11  -- ^^^^^^^^^^ allocating an object,
12  --             which becomes the designated
13  --             object for I1
14
15 end Show_Simple_Allocation;
```

Code block metadata

Project: `Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Simple_Allocation`
MD5: `32ca8cf523e19b25dabb55da6df1f18d`

In this example, `I1` is an access object and the object allocated via **new Integer** is its designated object.

Access value and designated value

An access object and a designated (allocated) object, both store values. The value of an access object is the *access value* and the value of a designated object is the *designated value*. For example:

Listing 1277: show_values.adb

```

1  procedure Show_Values is
2
3      -- Access-to-object type
4      type Integer_Access is access Integer;
5
6      I1, I2, I3 : Integer_Access;
7
8  begin
9      I1 := new Integer;
10     I3 := new Integer;
11
12     -- Copying the access value of I1 to I2
13     I2 := I1;
14
15     -- Copying the designated value of I1
16     I3.all := I1.all;
17
18 end Show_Values;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Terminology.Values
MD5: a152ee813b8ed9fad985cf4e2c25d847

In this example, the assignment `I2 := I1` copies the access value of `I1` to `I2`. The assignment `I3.all := I1.all` copies `I1`'s designated value to `I3`'s designated object. (As we already know, `.all` is used to dereference an access object. We discuss this topic again *later in this chapter* (page 911).)

** In the Ada Reference Manual**

- 3.10 Access Types²⁴²

28.1.2 Access types: Allocation

Ada makes the distinction between pool-specific and general access types, as we'll discuss in this section. Before doing so, however, let's talk about memory allocation.

In general terms, memory can be allocated dynamically on the heap or statically on the stack. (Strictly speaking, both are dynamic allocations, in that they occur at run-time with amounts not previously specified.) For example:

Listing 1278: show_simple_allocation.adb

```

1  procedure Show_Simple_Allocation is
2
3      -- Declaring access type:
4      type Integer_Access is access Integer;
5
6      -- Declaring access object:
7      A1 : Integer_Access;
8
9  begin
10     -- Allocating an Integer object on the heap
11     A1 := new Integer;
12

```

(continues on next page)

²⁴² <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

13  declare
14      -- Allocating an Integer object on the
15      -- stack
16      I : Integer;
17  begin
18      null;
19  end;
20
21  end Show_Simple_Allocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↳ Allocation.Simple_Allocation
MD5: 4144feb99e6e0b1a0749fce0b20370a1

When we allocate an object on the heap via **new**, the allocation happens in a memory pool that is associated with the access type. In our code example, there's a memory pool associated with the `Integer_Access` type, and each **new Integer** allocates a new integer object in that pool. Therefore, access types of this kind are called pool-specific access types. (We discuss *more about these types* (page 885) later.)

It is also possible to access objects that were allocated on the stack. To do that, however, we cannot use pool-specific access types because — as the name suggests — they're only allowed to access objects that were allocated in the specific pool associated with the type. Instead, we have to use general access types in this case:

Listing 1279: show_general_access_type.adb

```

1  procedure Show_General_Access_Type is
2
3      -- Declaring general access type:
4      type Integer_Access is access all Integer;
5
6      -- Declaring access object:
7      A1 : Integer_Access;
8
9      -- Allocating an Integer object on the
10     -- stack:
11     I : aliased Integer;
12
13  begin
14      -- Getting access to an Integer object that
15      -- was allocated on the stack
16      A1 := I'Access;
17
18  end Show_General_Access_Type;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
↳ Allocation.General_Access_Types
MD5: f166291ad1975396131775d0aff6ad9d

In this example, we declare the general access type `Integer_Access` and the access object `A1`. To initialize `A1`, we write `I'Access` to get access to an integer object `I` that was allocated on the stack. (For the moment, don't worry much about these details: we'll talk about general access types again when we introduce the topic of *aliased objects* (page 924) later on.)

i For further reading...

Note that it is possible to use general access types to allocate objects on the heap:

Listing 1280: show_simple_allocation.adb

```

1  procedure Show_Simple_Allocation is
2
3      -- Declaring general access type:
4      type Integer_Access is access all Integer;
5
6      -- Declaring access object:
7      A1 : Integer_Access;
8
9  begin
10     --
11     -- Allocating an Integer object on the heap
12     -- and initializing an access object of
13     -- the general access type Integer_Access.
14     --
15     A1 := new Integer;
16
17 end Show_Simple_Allocation;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
Allocation.General_Access_Types_Heap
MD5: 3fa5efeac2f66794f066ab29f26bf7ca

Here, we're using a general access type `Integer_Access`, but allocating an integer object on the heap.

i Important

In many code examples, we have used the `Integer` type as the designated subtype of the access types — by writing `access Integer`. Although we have used this specific scalar type, we aren't really limited to those types. In fact, we can use *any type* as the designated subtype, including user-defined types, composite types, task types and protected types.

i In the Ada Reference Manual

- [3.10 Access Types](#)²⁴³

Pool-specific access types

We've already discussed many aspects about pool-specific access types. In this section, we recapitulate some of those aspects, and discuss some new details that haven't seen yet.

As we know, we cannot directly assign an object `Distance_Miles` of type `Miles` to an object `Distance_Meters` of type `Meters`, even if both share a common `Float` type ancestor. The assignment is only possible if we perform a type conversion from `Miles` to `Meters`, or vice-versa — e.g.: `Distance_Meters := Meters (Distance_Miles) * Miles_To_Meters_Factor`.

²⁴³ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Similarly, in the case of pool-specific access types, a direct assignment between objects of different access types isn't possible. However, even if both access types have the same designated subtype (let's say, they are both declared using **is access Integer**), it's still not possible to perform a type conversion between those access types. The only situation when an access type conversion is allowed is when both types have a common ancestor.

Let's see an example:

Listing 1281: show_simple_allocation.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Allocation is
4
5     -- Declaring pool-specific access type:
6     type Integer_Access_1 is access Integer;
7     type Integer_Access_2 is access Integer;
8     type Integer_Access_2B is new Integer_Access_2;
9
10    -- Declaring access object:
11    A1 : Integer_Access_1;
12    A2 : Integer_Access_2;
13    A2B : Integer_Access_2B;
14
15 begin
16     A1 := new Integer;
17     Put_Line ("A1 : " & A1'Image);
18     Put_Line ("Pool: " & A1'Storage_Pool'Image);
19
20     A2 := new Integer;
21     Put_Line ("A2:  " & A2'Image);
22     Put_Line ("Pool: " & A2'Storage_Pool'Image);
23
24     -- ERROR: Cannot directly assign access values
25     --         for objects of unrelated access
26     --         types; also, cannot convert between
27     --         these types.
28     --
29     -- A1 := A2;
30     -- A1 := Integer_Access_1 (A2);
31
32     A2B := Integer_Access_2B (A2);
33     Put_Line ("A2B: " & A2B'Image);
34     Put_Line ("Pool: " & A2B'Storage_Pool'Image);
35
36 end Show_Simple_Allocation;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
Allocation.Pool_Specific_Access_Types
MD5: 80d0e9764917fa8352b6616e3a8425de

Runtime output

```
A1 : (access 584c2d4a92a0)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL' {SYSTEM.STORAGE_POOLS.TROOT_
  ↳ STORAGE_POOLC object}
A2:  (access 584c2d4a9360)
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL' {SYSTEM.STORAGE_POOLS.TROOT_
  ↳ STORAGE_POOLC object}
A2B: (access 584c2d4a9360)
```

(continues on next page)

(continued from previous page)

```
Pool: SYSTEM.POOL_GLOBAL.UNBOUNDED_NO_RECLAIM_POOL' {SYSTEM.STORAGE_POOLS.TROOT_
  ↳ STORAGE_POOLC object}
```

In this example, we declare three access types: `Integer_Access_1`, `Integer_Access_2` and `Integer_Access_2B`. Also, the `Integer_Access_2B` type is derived from the `Integer_Access_2` type. Therefore, we can convert an object of `Integer_Access_2` type to the `Integer_Access_2B` type — we do this in the `A2B := Integer_Access_2B (A2)` assignment. However, we cannot directly assign to or convert between unrelated types such as `Integer_Access_1` and `Integer_Access_2`. (We would get a compilation error if we included the `A1 := A2` or the `A1 := Integer_Access_1 (A2)` assignment.)

Important

Remember that:

- As mentioned in the [Introduction to Ada course](#) (page 97):
 - an access type can be unconstrained, but the actual object allocation must be constrained;
 - we can use a [qualified expression](#) (page 338) to allocate an object.
- We can use the `Storage_Size` attribute to limit the size of the memory pool associated with an access type, as discussed previously in the [section about storage size](#) (page 359).
- When running out of memory while allocating via **new**, we get a `Storage_Error` exception because of the [storage check](#) (page 822).

For example:

Listing 1282: `show_array_allocation.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Array_Allocation is
4
5   -- Unconstrained array type:
6   type Integer_Array is
7     array (Positive range <>) of Integer;
8
9   -- Access type with unconstrained
10  -- designated subtype and limited storage
11  -- size.
12  type Integer_Array_Access is
13    access Integer_Array
14    with Storage_Size => 128;
15
16  -- An access object:
17  A1 : Integer_Array_Access;
18
19  procedure Show_Info
20    (IAA : Integer_Array_Access) is
21  begin
22    Put_Line ("Allocated: " & IAA'Image);
23    Put_Line ("Length:      "
24              & IAA.all'Length'Image);
25    Put_Line ("Values:      "
26              & IAA.all'Image);
27  end Show_Info;
28
29 begin
```

```

30  -- Allocating an integer array with
31  -- constrained range on the heap:
32  A1 := new Integer_Array (1 .. 3);
33  A1.all := [others => 42];
34  Show_Info (A1);
35
36  -- Allocating an integer array on the
37  -- heap using a qualified expression:
38  A1 := new Integer_Array'(5, 10);
39  Show_Info (A1);
40
41  -- A third allocation fails at run time
42  -- because of the constrained storage
43  -- size:
44  A1 := new Integer_Array (1 .. 100);
45  Show_Info (A1);
46
47  exception
48  when Storage_Error =>
49      Put_Line ("Out of memory!");
50
51  end Show_Array_Allocation;

```

Multiple allocation

Up to now, we have seen examples of allocating a single object on the heap. It's possible to allocate multiple objects *at once* as well — i.e. syntactic sugar is available to simplify the code that performs this allocation. For example:

Listing 1283: show_access_array_allocation.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Access_Array_Allocation is
4
5      type Integer_Access is access Integer;
6
7      type Integer_Access_Array is
8          array (Positive range <>) of Integer_Access;
9
10     -- An array of access objects:
11     Arr : Integer_Access_Array (1 .. 10);
12
13  begin
14     --
15     -- Allocating 10 access objects and
16     -- initializing the corresponding designated
17     -- object with zero:
18     --
19     Arr := (others => new Integer'(0));
20
21     -- Same as:
22     for I in Arr'Range loop
23         Arr (I) := new Integer'(0);
24     end loop;
25
26     Put_Line ("Arr: " & Arr'Image);
27
28     Put_Line ("Arr (designated values): ");
29     for E of Arr loop

```

(continues on next page)

(continued from previous page)

```

30     Put (E.all'Image);
31 end loop;
32
33 end Show_Access_Array_Allocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
Allocation.Integer_Access_Array
MD5: 4afc9358c8aa9426a97ca8932c75d932

Runtime output

```

Arr:
[(access 56d3a7a8d3e0), (access 56d3a7a8d400), (access 56d3a7a8d420),
 (access 56d3a7a8d440), (access 56d3a7a8d460), (access 56d3a7a8d480),
 (access 56d3a7a8d4a0), (access 56d3a7a8d4c0), (access 56d3a7a8d4e0),
 (access 56d3a7a8d500)]
Arr (designated values):
0 0 0 0 0 0 0 0 0 0

```

In this example, we have the access type `Integer_Access` and an array type of this access type (`Integer_Access_Array`). We also declare an array `Arr` of `Integer_Access_Array` type. This means that each component of `Arr` is an access object. We allocate all ten components of the `Arr` array by simply writing `Arr := (others => new Integer)`. This *array aggregate* (page 531) is syntactic sugar for a loop over `Arr` that allocates each component. (Note that, by writing `Arr := (others => new Integer' (0))`, we're also initializing the designated objects with zero.)

Let's see another code example, this time with task types:

Listing 1284: workers.ads

```

1 package Workers is
2
3     task type Worker is
4         entry Start (Id : Positive);
5         entry Stop;
6     end Worker;
7
8     type Worker_Access is access Worker;
9
10    type Worker_Array is
11        array (Positive range <>) of Worker_Access;
12
13 end Workers;

```

Listing 1285: workers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Workers is
4
5     task body Worker is
6         Id : Positive;
7     begin
8         accept Start (Id : Positive) do
9             Worker.Id := Id;
10        end Start;
11        Put_Line ("Started Worker #")

```

(continues on next page)

(continued from previous page)

```
12         & Id'Image);
13
14     accept Stop;
15
16     Put_Line ("Stopped Worker #"
17             & Id'Image);
18 end Worker;
19
20 end Workers;
```

Listing 1286: show_workers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Workers; use Workers;
4
5 procedure Show_Workers is
6     Worker_Arr : Worker_Array (1 .. 20);
7 begin
8     --
9     --   Allocating 20 workers at once:
10    --
11    Worker_Arr := (others => new Worker);
12
13    for I in Worker_Arr'Range loop
14        Worker_Arr (I).Start (I);
15    end loop;
16
17    Put_Line ("Some processing...");
18    delay 1.0;
19
20    for W of Worker_Arr loop
21        W.Stop;
22    end loop;
23
24 end Show_Workers;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Types_
Allocation.Workers
MD5: d29e3d56585f8d9a63b805c680e5dc54

Runtime output

```
Started Worker # 1
Started Worker # 2
Started Worker # 3
Started Worker # 4
Started Worker # 5
Started Worker # 6
Started Worker # 7
Started Worker # 8
Started Worker # 9
Started Worker # 10
Started Worker # 11
Started Worker # 12
Started Worker # 13
Started Worker # 14
Started Worker # 15
Started Worker # 16
```

(continues on next page)

(continued from previous page)

```

Started Worker # 17
Started Worker # 18
Started Worker # 19
Started Worker # 20
Some processing...
Stopped Worker # 1
Stopped Worker # 2
Stopped Worker # 3
Stopped Worker # 5
Stopped Worker # 4
Stopped Worker # 8
Stopped Worker # 7
Stopped Worker # 6
Stopped Worker # 9
Stopped Worker # 10
Stopped Worker # 11
Stopped Worker # 12
Stopped Worker # 14
Stopped Worker # 13
Stopped Worker # 15
Stopped Worker # 16
Stopped Worker # 17
Stopped Worker # 18
Stopped Worker # 19
Stopped Worker # 20

```

In this example, we declare the task type `Worker`, the access type `Worker_Access` and an array of access to tasks `Worker_Array`. Using this approach, a task is only created when we allocate an individual component of an array of `Worker_Array` type. Thus, when we declare the `Worker_Arr` array in this example, we're only preparing a *container* of 20 workers, but we don't have any actual tasks yet. We bring the 20 tasks into existence by writing `Worker_Arr := (others => new Worker)`.

28.1.3 Discriminants as Access Values

We can use access types when declaring discriminants. Let's see an example:

Listing 1287: custom_recs.ads

```

1 package Custom_Recs is
2
3   -- Declaring an access type:
4   type Integer_Access is access Integer;
5
6   -- Declaring a discriminant with this
7   -- access type:
8   type Rec (IA : Integer_Access) is record
9
10      I : Integer := IA.all;
11      -- ^^^^^^^
12      -- Setting I's default to use the
13      -- designated value of IA:
14   end record;
15
16   procedure Show (R : Rec);
17
18 end Custom_Recs;

```

Listing 1288: custom_recs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Recs is
4
5     procedure Show (R : Rec) is
6     begin
7         Put_Line ("R.IA = "
8                 & Integer'Image (R.IA.all));
9         Put_Line ("R.I = "
10                & Integer'Image (R.I));
11     end Show;
12
13 end Custom_Recs;

```

Listing 1289: show_discriminants_as_access_values.adb

```

1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Discriminants_As_Access_Values is
4
5     IA : constant Integer_Access :=
6         new Integer'(10);
7     R : Rec (IA);
8
9     begin
10        Show (R);
11
12        IA.all := 20;
13        R.I := 30;
14        Show (R);
15
16        -- As expected, we cannot change the
17        -- discriminant. The following line is
18        -- triggers a compilation error:
19        --
20        -- R.IA := new Integer;
21
22 end Show_Discriminants_As_Access_Values;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_Access_Values.Discriminant_Access_Values
 MD5: c7850acefd8e5227f4be654faed13055

Runtime output

```

R.IA = 10
R.I = 10
R.IA = 20
R.I = 30

```

In the Custom_Recs package from this example, we declare the access type Integer_Access. We then use this type to declare the discriminant (IA) of the Rec type. In the Show_Discriminants_As_Access_Values procedure, we see that (as expected) we cannot change the discriminant of an object of Rec type: an assignment such as R.IA := new Integer would trigger a compilation error.

Note that we can use a default for the discriminant:

Listing 1290: custom_recs.ads

```

1 package Custom_Recs is
2
3     type Integer_Access is access Integer;
4
5     type Rec (IA : Integer_Access
6               := new Integer'(0)) is
7         --
8         --             default value
9     record
10         I : Integer := IA.all;
11     end record;
12
13     procedure Show (R : Rec);
14
15 end Custom_Recs;

```

Listing 1291: show_discriminants_as_access_values.adb

```

1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Discriminants_As_Access_Values is
4
5     R1 : Rec;
6     --   ^^
7     --   no discriminant: use default
8
9     R2 : Rec (new Integer'(20));
10    --   ^^^^^^^^^^^^^^^^^
11    --   allocating an unnamed integer object
12
13 begin
14     Show (R1);
15     Show (R2);
16 end Show_Discriminants_As_Access_Values;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_Access_Values.Discriminant_Access_Values
MD5: 968cb88ed7e9e6958ab66fb6f5a7ce2d

Runtime output

```

R.IA = 0
R.I  = 0
R.IA = 20
R.I  = 20

```

Here, we've changed the declaration of the Rec type to allocate an integer object if the type's discriminant isn't provided — we can see this in the declaration of the R1 object in the Show_Discriminants_As_Access_Values procedure. Also, in this procedure, we're allocating an unnamed integer object in the declaration of R2.

In the Ada Reference Manual

- 3.10 Access Types²⁴⁴
- 3.7.1 Discriminant Constraints²⁴⁵

Unconstrained type as designated subtype

Notice that we were using a scalar type as the designated subtype of the `Integer_Access` type. We could have used an unconstrained type as well. In fact, this is often used for the sake of having the effect of an unconstrained discriminant type.

Let's see an example:

Listing 1292: persons.ads

```
1 package Persons is
2
3   -- Declaring an access type whose
4   -- designated subtype is unconstrained:
5   type String_Access is access String;
6
7   -- Declaring a discriminant with this
8   -- access type:
9   type Person (Name : String_Access) is record
10     Age : Integer;
11   end record;
12
13   procedure Show (P : Person);
14
15 end Persons;
```

Listing 1293: persons.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Persons is
4
5   procedure Show (P : Person) is
6   begin
7     Put_Line ("Name = "
8              & P.Name.all);
9     Put_Line ("Age = "
10             & Integer'Image (P.Age));
11   end Show;
12
13 end Persons;
```

Listing 1294: show_person.adb

```
1 with Persons; use Persons;
2
3 procedure Show_Person is
4   P : Person (new String'("John"));
5 begin
6   P.Age := 30;
7   Show (P);
8 end Show_Person;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
Access_Values.Persons
MD5: 9b1109d076b6f06632c8685a41616210

Runtime output

²⁴⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

²⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-7-1.html>

```
Name = John
Age  = 30
```

In this example, the discriminant of the Person type has an unconstrained designated type. In the Show_Person procedure, we declare the P object and specify the constraints of the allocated string object — in this case, a four-character string initialized with the name "John".

For further reading...

In the previous code example, we used an array — actually, a string — to demonstrate the advantage of using discriminants as access values, for we can use an unconstrained type as the designated subtype. In fact, as we discussed [earlier in another chapter](#) (page 467), we can only use discrete types (or access types) as discriminants. Therefore, you wouldn't be able to use a string, for example, directly as a discriminant without using access types:

Listing 1295: persons.ads

```
1 package Persons is
2
3   -- ERROR: Declaring a discriminant with an
4   --           unconstrained type:
5   type Person (Name : String) is record
6     Age : Integer;
7   end record;
8
9 end Persons;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
 ↪ Access_Values.Persons_Error
 MD5: 4144852aaf95da62bc4781b1e8dc2717

Build output

```
persons.ads:5:24: error: discriminants must have a discrete or access type
gprbuild: *** compilation phase failed
```

As expected, compilation fails for this code because the discriminant of the Person type is indefinite.

However, the advantage of discriminants as access values isn't restricted to being able to use unconstrained types such as arrays: we could really use any type as the designated subtype! In fact, we can generalize this to:

Listing 1296: gen_custom_recs.ads

```
1 generic
2   type T (<>); -- any type
3   type T_Access is access T;
4   package Gen_Custom_Recs is
5     -- Declare a type whose discriminant D can
6     -- access any type:
7     type T_Rec (D : T_Access) is null record;
8   end Gen_Custom_Recs;
```

Listing 1297: custom_recs.ads

```

1 with Gen_Custom_Recs;
2
3 package Custom_Recs is
4
5     type Incomp;
6     -- Incomplete type declaration!
7
8     type Incomp_Access is access Incomp;
9
10    -- Instantiating package using
11    -- incomplete type Incomp:
12    package Inst is new
13        Gen_Custom_Recs
14        (T => Incomp,
15         T_Access => Incomp_Access);
16    subtype Rec is Inst.T_Rec;
17
18    -- At this point, Rec (Inst.T_Rec) uses
19    -- an incomplete type as the designated
20    -- subtype of its discriminant type
21
22    procedure Show (R : Rec) is null;
23
24    -- Now, we complete the Incomp type:
25    type Incomp (B : Boolean := True) is private;
26
27 private
28     -- Finally, we have the full view of the
29     -- Incomp type:
30     type Incomp (B : Boolean := True) is
31         null record;
32
33 end Custom_Recs;

```

Listing 1298: show_rec.adb

```

1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Rec is
4     R : Rec (new Incomp);
5 begin
6     Show (R);
7 end Show_Rec;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_Access_Values.Generic_Access
 MD5: c65510e8c6a7625cbd08aa9e68f05115

In the Gen_Custom_Recs package, we're using **type T** (<>) — which can be any type — for the designated subtype of the access type T_Access, which is the type of T_Rec's discriminant. In the Custom_Recs package, we use the incomplete type Incomp to instantiate the generic package. Only after the instantiation, we declare the complete type.

Later on, we'll discuss discriminants again when we look into *anonymous access discriminants* (page 1011), which provide some advantages in terms of *accessibility rules* (page 932).

Whole object assignments

As expected, we cannot change the discriminant value in whole object assignments. If we do that, the `Constraint_Error` exception is raised at runtime:

Listing 1299: show_person.adb

```

1  with Persons; use Persons;
2
3  procedure Show_Person is
4      S1 : String_Access := new String'("John");
5      S2 : String_Access := new String'("Mark");
6      P : Person := (Name => S1,
7                      Age  => 30);
8  begin
9      P := (Name => S1, Age => 31);
10         --      ^^ OK: we didn't change the
11         --      discriminant.
12     Show (P);
13
14     -- We can just repeat the discriminant:
15     P := (Name => P.Name, Age => 32);
16     --      ^^^^^^ OK: we didn't change the
17     --      discriminant.
18     Show (P);
19
20     -- Of course, we can change the string itself:
21     S1.all := "Mark";
22     Show (P);
23
24     P := (Name => S2, Age => 40);
25     --      ^^ ERROR: we changed the
26     --      discriminant!
27     Show (P);
28 end Show_Person;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Discriminants_As_
 ↪ Access_Values.Persons
 MD5: 96f4742365eb6a07c377a5dec28b5767

Runtime output

```

Name = John
Age  = 31
Name = John
Age  = 32
Name = Mark
Age  = 32

```

```
raised CONSTRAINT_ERROR : show_person.adb:24 discriminant check failed
```

The first and the second assignments to `P` are OK because we didn't change the discriminant. However, the last assignment raises the `Constraint_Error` exception at runtime because we're changing the discriminant.

28.1.4 Parameters as Access Values

In addition to *using discriminants as access values* (page 891), we can use access types for subprogram formal parameters. For example, the N parameter of the Show procedure below has an access type:

Listing 1300: names.ads

```
1 package Names is
2
3   type Name is access String;
4
5   procedure Show (N : Name);
6
7 end Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↳ Access_Values.Names
MD5: 82ce94987dce9026aed54a0deb3cc548

This is the complete code example:

Listing 1301: names.ads

```
1 package Names is
2
3   type Name is access String;
4
5   procedure Show (N : Name);
6
7 end Names;
```

Listing 1302: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5   procedure Show (N : Name) is
6   begin
7     Put_Line ("Name: " & N.all);
8   end Show;
9
10 end Names;
```

Listing 1303: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4   N : Name := new String'("John");
5 begin
6   Show (N);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↳ Access_Values.Names
MD5: 526baf1996b4a2970c3fa2e3485dcbad

Runtime output

Name: John

Note that in this example, the Show procedure is basically just displaying the string. Since the procedure isn't doing anything that justifies the need for an access type, we could have implemented it with a *simpler* type:

Listing 1304: names.ads

```

1 package Names is
2
3     type Name is access String;
4
5     procedure Show (N : String);
6
7 end Names;
```

Listing 1305: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5     procedure Show (N : String) is
6     begin
7         Put_Line ("Name: " & N);
8     end Show;
9
10 end Names;
```

Listing 1306: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := new String' ("John");
5 begin
6     Show (N.all);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
 ↪ Access_Values.Names_String
 MD5: 097ec1ff781fda9deed1de23cae39ae5

Runtime output

Name: John

It's important to highlight the difference between passing an access value to a subprogram and passing an object by reference. In both versions of this code example, the compiler will make use of a reference for the actual parameter of the N parameter of the Show procedure. However, the difference between these two cases is that:

- N : Name is a reference to an object (because it's an access value) that is passed by value, and
- N : String is an object passed by reference.

Changing the referenced object

Since the Name type gives us access to an object in the Show procedure, we could actually change this object inside the procedure. To illustrate this, let's change the Show procedure to lower each character of the string before displaying it (and rename the procedure to Lower_And_Show):

Listing 1307: names.ads

```
1 package Names is
2
3     type Name is access String;
4
5     procedure Lower_And_Show (N : Name);
6
7 end Names;
```

Listing 1308: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Lower_And_Show (N : Name) is
9     begin
10         for I in N'Range loop
11             N (I) := To_Lower (N (I));
12         end loop;
13         Put_Line ("Name: " & N.all);
14     end Lower_And_Show;
15
16 end Names;
```

Listing 1309: show_changed_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4     N : Name := new String ("John");
5 begin
6     Lower_And_Show (N);
7 end Show_Changed_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
Access_Values.Changed_Names
MD5: 063a507284f5e7ffa669db2c8fdd3d6f

Runtime output

```
Name: john
```

Notice that, again, we could have implemented the Lower_And_Show procedure without using an access type:

Listing 1310: names.ads

```

1 package Names is
2
3     type Name is access String;
4
5     procedure Lower_And_Show (N : in out String);
6
7 end Names;
```

Listing 1311: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use   Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Lower_And_Show (N : in out String) is
9     begin
10         for I in N'Range loop
11             N (I) := To_Lower (N (I));
12         end loop;
13         Put_Line ("Name: " & N);
14     end Lower_And_Show;
15
16 end Names;
```

Listing 1312: show_changed_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4     N : Name := new String'("John");
5 begin
6     Lower_And_Show (N.all);
7 end Show_Changed_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↳ Access_Values.Changed_Names_String
MD5: 783ea8c45ed8ad3e0007524c11b6bcc4

Runtime output

Name: john

Replace the access value

Instead of changing the object in the Lower_And_Show procedure, we could replace the access value by another one — for example, by allocating a new string inside the procedure. In this case, we have to pass the access value by reference using the **in out** parameter mode:

Listing 1313: names.ads

```

1 package Names is
2
```

(continues on next page)

(continued from previous page)

```
3   type Name is access String;
4
5   procedure Lower_And_Show (N : in out Name);
6
7 end Names;
```

Listing 1314: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Ada.Characters.Handling;
4 use   Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Lower_And_Show (N : in out Name) is
9   begin
10      N := new String'(To_Lower (N.all));
11      Put_Line ("Name: " & N.all);
12   end Lower_And_Show;
13
14 end Names;
```

Listing 1315: show_changed_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Changed_Names is
4   N : Name := new String'("John");
5 begin
6   Lower_And_Show (N);
7 end Show_Changed_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
↳ Access_Values.Replaced_Names
MD5: a4abfe6fdb1e5029e8eea17641cd960b

Runtime output

Name: john

Now, instead of changing the object referenced by N, we're actually replacing it with a new object that we allocate inside the Lower_And_Show procedure.

As expected, contrary to the previous examples, we cannot implement this code by relying on parameter modes to replace the object. In fact, we have to use access types for this kind of operations.

Note that this implementation creates a memory leak. In a proper implementation, we should make sure to *deallocate the object* (page 944), as explained later on.

Side-effects on designated objects

In previous code examples from this section, we've seen that passing a parameter by reference using the **in** or **in out** parameter modes is an alternative to using access values as parameters. Let's focus on the subprogram declarations of those code examples and their parameter modes:

Subprogram	Parameter type	Parameter mode
Show	Name	in
Show	String	in
Lower_And_Show	Name	in
Lower_And_Show	String	in out

When we analyze the information from this table, we see that in the case of using strings with different parameter modes, we have a clear indication whether the subprogram might change the object or not. For example, we know that a call to Show (N : String) won't change the string object that we're passing as the actual parameter.

In the case of passing an access value, we cannot know whether the designated object is going to be altered by a call to the subprogram. In fact, in both Show and Lower_And_Show procedures, the parameter is the same: N : Name — in other words, the parameter mode is in in both cases. Here, there's no clear indication about the effects of a subprogram call on the designated object.

The simplest way to ensure that the object isn't changed in the subprogram is by using *access-to-constant types* (page 925), which we discuss later on. In this case, we're basically saying that the object we're accessing in Show is constant, so we cannot possibly change it:

Listing 1316: names.ads

```

1 package Names is
2
3     type Name is access String;
4
5     type Constant_Name is access constant String;
6
7     procedure Show (N : Constant_Name);
8
9 end Names;
```

Listing 1317: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : Constant_Name) is
9     begin
10         -- for I in N'Range loop
11             N (I) := To_Lower (N (I));
12         -- end loop;
13         Put_Line ("Name: " & N.all);
14     end Show;
15
16 end Names;
```

Listing 1318: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := new String'("John");
```

(continues on next page)

(continued from previous page)

```

5 begin
6   Show (Constant_Name (N));
7 end Show_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
 ↳ Access_Values.Names_Constant
 MD5: 77526e0a159bf1bcbef08a21be250f3c

Runtime output

Name: John

In this case, the `Constant_Name` type ensures that the `N` parameter won't be changed in the `Show` procedure. Note that we need to convert from `Name` to `Constant_Name` to be able to call the `Show` procedure (in the `Show_Names` procedure). Although using `in String` is still a simpler solution, this approach works fine.

(Feel free to uncomment the call to `To_Lower` in the `Show` procedure and the corresponding with- and use-clauses to see that the compilation fails when trying to change the constant object.)

We could also mitigate the problem by using contracts. For example:

Listing 1319: names.ads

```

1 package Names is
2
3   type Name is access String;
4
5   procedure Show (N : Name)
6     with Post => N.all'Old = N.all;
7   --      ~~~~~
8   --      we promise that we won't change
9   --      the object
10
11 end Names;

```

Listing 1320: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Show (N : Name) is
9   begin
10    -- for I in N'Range loop
11    --   N (I) := To_Lower (N (I));
12    -- end loop;
13    Put_Line ("Name: " & N.all);
14   end Show;
15
16 end Names;

```

Listing 1321: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4   N : Name := new String'("John");
5 begin
6   Show (N);
7 end Show_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
 ↪ Access_Values.Names_Postcondition
 MD5: 2a70993232baca9d58d36e537a6fd32b

Runtime output

Name: John

Although a bit more verbose than a simple `in String`, the information in the specification of `Show` at least gives us an indication that the object won't be affected by the call to this subprogram. Note that this code actually compiles if we try to modify `N.all` in the `Show` procedure, but the post-condition fails at runtime when we do that.

(By uncommentating and building the code again, you'll see an exception being raised at runtime when trying to change the object.)

In the postcondition above, we're using `'Old` to refer to the original object before the subprogram call. Unfortunately, we cannot use this attribute when dealing with *limited private types* (page 1071) — or limited types in general. For example, let's change the declaration of `Name` and have it as a limited private type instead:

Listing 1322: names.ads

```

1 package Names is
2
3   type Name is limited private;
4
5   function Init (S : String) return Name;
6
7   function Equal (N1, N2 : Name)
8     return Boolean;
9
10  procedure Show (N : Name)
11    with Post => Equal (N'Old = N);
12
13 private
14
15   type Name is access String;
16
17   function Init (S : String) return Name is
18     (new String'(S));
19
20   function Equal (N1, N2 : Name)
21     return Boolean is
22     (N1.all = N2.all);
23
24 end Names;

```

Listing 1323: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8     procedure Show (N : Name) is
9     begin
10         -- for I in N'Range loop
11         --     N (I) := To_Lower (N (I));
12         -- end loop;
13         Put_Line ("Name: " & N.all);
14     end Show;
15
16 end Names;
```

Listing 1324: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     N : Name := Init ("John");
5 begin
6     Show (N);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
 ↪ Access_Values.Names_Limited_Private
 MD5: 39691394d7a934869dc569eb72d1bf3a

Build output

```

names.ads:11:26: error: attribute "Old" cannot apply to limited objects
gprbuild: *** compilation phase failed
```

In this case, we have no means to indicate that a call to Show won't change the internal state of the actual parameter.

For further reading...

As an alternative, we could declare a new `Constant_Name` type that is also limited private. If we use this type in Show procedure, we're at least indicating (in the type name) that the type is supposed to be constant — even though we're not directly providing means to actually ensure that no modifications occur in a call to the procedure. However, the fact that we declare this type as an access-to-constant (in the private part of the specification) makes it clear that a call to Show won't change the designated object.

Let's look at the adapted code:

Listing 1325: names.ads

```

1 package Names is
2
3     type Name is limited private;
4
5     type Constant_Name is limited private;
```

```

6
7   function Init (S : String) return Name;
8
9   function To_Constant_Name
10     (N : Name)
11     return Constant_Name;
12
13   procedure Show (N : Constant_Name);
14
15 private
16
17   type Name is
18     access String;
19
20   type Constant_Name is
21     access constant String;
22
23   function Init (S : String) return Name is
24     (new String'(S));
25
26   function To_Constant_Name
27     (N : Name)
28     return Constant_Name is
29     (Constant_Name (N));
30
31 end Names;

```

Listing 1326: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use  Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Show (N : Constant_Name) is
9   begin
10     -- for I in N'Range loop
11     --   N (I) := To_Lower (N (I));
12     -- end loop;
13     Put_Line ("Name: " & N.all);
14   end Show;
15
16 end Names;

```

Listing 1327: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4   N : Name := Init ("John");
5 begin
6   Show (To_Constant_Name (N));
7 end Show_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Parameters_As_
 Access_Values.Names_Constant_Limited_Private
 MD5: 30da588b57e6b4dfbf9934f77d348473

Runtime output

Name: John

In this version of the source code, the Show procedure doesn't have any side-effects, as we cannot modify N inside the procedure.

Having the information about the effects of a subprogram call to an object is very important: we can use this information to set expectations — and avoid unexpected changes to an object. Also, this information can be used to prove that a program works as expected. Therefore, whenever possible, we should avoid access values as parameters. Instead, we can rely on appropriate parameter modes and pass an object by reference.

There are cases, however, where the design of our application doesn't permit replacing the access type with simple parameter modes. Whenever we have an abstract data type encapsulated as a limited private type — such as in the last code example —, we might have no means to avoid access values as parameters. In this case, using the access type is of course justifiable. We'll see such a case in the *next section* (page 908).

28.1.5 Self-reference

As we've discussed in the section about incomplete types <Adv_Ada_Incomplete_Types>, we can use incomplete types to create a recursive, self-referencing type. Let's revisit a code example from that section:

Listing 1328: linked_list_example.ads

```
1 package Linked_List_Example is
2
3     type Integer_List;
4
5     type Next is access Integer_List;
6
7     type Integer_List is record
8         I : Integer;
9         N : Next;
10    end record;
11
12 end Linked_List_Example;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Self_Reference.
↳ Linked_List_Example
MD5: b2d3a048473d498bbe691bc6e38ca1e9

Here, we're using the incomplete type Integer_List in the declaration of the Next type, which we then use in the complete declaration of the Integer_List type.

Self-references are useful, for example, to create unbounded containers — such as the linked lists mentioned in the example above. Let's extend this code example and partially implement a generic package for linked lists:

Listing 1329: linked_lists.ads

```
1 generic
2     type T is private;
3 package Linked_Lists is
4
5     type List is limited private;
6
7     procedure Append_Front
8         (L : in out List;
9          E : T);
```

(continues on next page)

(continued from previous page)

```

10
11 procedure Append_Rear
12   (L : in out List;
13    E :      T);
14
15 procedure Show (L : List);
16
17 private
18
19   -- Incomplete type declaration:
20   type Component;
21
22   -- Using incomplete type:
23   type List is access Component;
24
25   type Component is record
26     Value : T;
27     Next  : List;
28     --    ^^^^
29     -- Self-reference via access type
30   end record;
31
32 end Linked_Lists;

```

Listing 1330: linked_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Linked_Lists is
4
5   procedure Append_Front
6     (L : in out List;
7      E :      T)
8   is
9     New_First : constant List := new
10       Component'(Value => E,
11                  Next  => L);
12   begin
13     L := New_First;
14   end Append_Front;
15
16   procedure Append_Rear
17     (L : in out List;
18      E :      T)
19   is
20     New_Last : constant List := new
21       Component'(Value => E,
22                  Next  => null);
23   begin
24     if L = null then
25       L := New_Last;
26     else
27       declare
28         Last : List := L;
29       begin
30         while Last.Next /= null loop
31           Last := Last.Next;
32         end loop;
33         Last.Next := New_Last;
34       end;
35     end if;

```

(continues on next page)

(continued from previous page)

```

36  end Append_Rear;
37
38  procedure Show (L : List) is
39      Curr : List := L;
40  begin
41      if L = null then
42          Put_Line ("[ ]");
43      else
44          Put ("[";
45          loop
46              Put (Curr.Value'Image);
47              Put (" ");
48              exit when Curr.Next = null;
49              Curr := Curr.Next;
50          end loop;
51          Put_Line ("]");
52      end if;
53  end Show;
54
55 end Linked_Lists;

```

Listing 1331: test_linked_list.adb

```

1  with Linked_Lists;
2
3  procedure Test_Linked_List is
4      package Integer_Lists is new
5          Linked_Lists (T => Integer);
6      use Integer_Lists;
7
8      L : List;
9  begin
10     Append_Front (L, 3);
11     Append_Rear (L, 4);
12     Append_Rear (L, 5);
13     Append_Front (L, 2);
14     Append_Front (L, 1);
15     Append_Rear (L, 6);
16     Append_Rear (L, 7);
17
18     Show (L);
19 end Test_Linked_List;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Self_Reference.
↳ Linked_List_Example
MD5: 6ab1f79c8c3e641eba8057874efc48d7

Runtime output

```
[ 1  2  3  4  5  6  7 ]
```

In this example, we declare an incomplete type Component in the private part of the generic Linked_Lists package. We use this incomplete type to declare the access type List, which is then used as a self-reference in the Next component of the Component type.

Note that we're using the List type *as a parameter* (page 898) for the Append_Front, Append_Rear and Show procedures.

In the Ada Reference Manual

- 3.10.1 Incomplete Type Declarations²⁴⁶

28.1.6 Mutually dependent types using access types

In the section on *mutually dependent types* (page 448), we've seen a code example where each type depends on the other one. We could rewrite that code example using access types:

Listing 1332: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T2;
4     type T2_Access is access T2;
5
6     type T1 is record
7         B : T2_Access;
8     end record;
9
10    type T1_Access is access T1;
11
12    type T2 is record
13        A : T1_Access;
14    end record;
15
16 end Mutually_Dependent;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Mutually_Dependent_Access_Types.Example
MD5: b21ffc4cdfe3db939dfc841cf8434344

In this example, T1 and T2 are mutually dependent types via the access types T1_Access and T2_Access — we're using those access types in the declaration of the B and A components.

28.1.7 Dereferencing

In the *Introduction to Ada course* (page 98), we discussed the `.all` syntax to dereference access values:

Listing 1333: show_dereferencing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     -- Declaring access type:
6     type Integer_Access is access Integer;
7
8     -- Declaring access object:
9     A1 : Integer_Access;
10
11 begin
```

(continues on next page)

²⁴⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-1.html>

(continued from previous page)

```
12  A1 := new Integer;
13
14  -- Dereferencing access value:
15  A1.all := 22;
16
17  Put_Line ("A1: " & Integer'Image (A1.all));
18  end Show_Dereferencing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.
↳ Simple_Dereferencing
MD5: 65655768c17a02991ffeda9a853b6ffb

Runtime output

```
A1:  22
```

In this example, we declare A1 as an access object, which allows us to access objects of **Integer** type. We dereference A1 by writing A1.all.

Here's another example, this time with an array:

Listing 1334: show_dereferencing.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Dereferencing is
4
5      type Integer_Array is
6          array (Positive range <>) of Integer;
7
8      type Integer_Array_Access is
9          access Integer_Array;
10
11     Arr : constant Integer_Array_Access :=
12           new Integer_Array (1 .. 6);
13  begin
14     Arr.all := (1, 2, 3, 5, 8, 13);
15
16     for I in Arr'Range loop
17         Put_Line ("Arr (: "
18                 & Integer'Image (I) & "): "
19                 & Integer'Image (Arr.all (I)));
20     end loop;
21  end Show_Dereferencing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_
↳ Dereferencing
MD5: 0e533dfd8ec1a74af17c99633c292e95

Runtime output

```
Arr (: 1):  1
Arr (: 2):  2
Arr (: 3):  3
Arr (: 4):  5
Arr (: 5):  8
Arr (: 6):  13
```

In this example, we dereference the access value by writing `Arr.all`. We then assign an array aggregate to it — this becomes `Arr.all := (... , ...)`;. Similarly, in the loop, we write `Arr.all (I)` to access the `I` component of the array.

In the Ada Reference Manual

- 4.1 Names²⁴⁷

Implicit Dereferencing

Implicit dereferencing allows us to omit the `.all` suffix without getting a compilation error. In this case, the compiler *knows* that the dereferenced object is implied, not the access value.

Ada supports implicit dereferencing in these use cases:

- when accessing components of a record or an array — including array slices.
- when accessing subprograms that have at least one parameter (we discuss this topic later in this chapter);
- when accessing some attributes — such as some array and task attributes.

Arrays

Let's start by looking into an example of implicit dereferencing of arrays. We can take the previous code example and replace `Arr.all (I)` by `Arr (I)`:

Listing 1335: show_dereferencing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Dereferencing is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     Arr : constant Integer_Array_Access :=
12           new Integer_Array (1 .. 6);
13 begin
14     Arr.all := (1, 2, 3, 5, 8, 13);
15
16     Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);
17
18     for I in Arr'Range loop
19         Put_Line
20             ("Arr (:"
21              & Integer'Image (I) & "): "
22              & Integer'Image (Arr (I)));
23         -- ^ .all is implicit.
24     end loop;
25 end Show_Dereferencing;
```

Code block metadata

²⁴⁷ <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_Implicit_Dereferencing
MD5: ade602a9e6976018e0c00f930a2399f1

Runtime output

```
Arr (: 1): 1
Arr (: 2): 2
Arr (: 3): 3
Arr (: 4): 5
Arr (: 5): 8
Arr (: 6): 13
```

Both forms — `Arr.all (I)` and `Arr (I)` — are equivalent. Note, however, that there's no implicit dereferencing when we want to access the whole array. (Therefore, we cannot write `Arr := (1, 2, 3, 5, 8, 13);`.) However, as slices are implicitly dereferenced, we can write `Arr (1 .. 6) := (1, 2, 3, 5, 8, 13);` instead of `Arr.all (1 .. 6) := (1, 2, 3, 5, 8, 13);`. Alternatively, we can assign to the array components individually and use implicit dereferencing for each component:

```
Arr (1) := 1;
Arr (2) := 2;
Arr (3) := 3;
Arr (4) := 5;
Arr (5) := 8;
Arr (6) := 13;
```

Implicit dereferencing isn't available for the whole array because we have to distinguish between assigning to access objects and assigning to actual arrays. For example:

Listing 1336: show_array_assignments.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Array_Assignments is
4
5     type Integer_Array is
6         array (Positive range <>) of Integer;
7
8     type Integer_Array_Access is
9         access Integer_Array;
10
11     procedure Show_Array
12         (Name : String;
13          Arr  : Integer_Array_Access) is
14     begin
15         Put (Name);
16         for E of Arr.all loop
17             Put (Integer'Image (E));
18         end loop;
19         New_Line;
20     end Show_Array;
21
22     Arr_1 : constant Integer_Array_Access :=
23         new Integer_Array (1 .. 6);
24     Arr_2 : Integer_Array_Access :=
25         new Integer_Array (1 .. 6);
26 begin
27     Arr_1.all := (1, 2, 3, 5, 8, 13);
28     Arr_2.all := (21, 34, 55, 89, 144, 233);
29
```

(continues on next page)

(continued from previous page)

```

30  -- Array assignment
31  Arr_2.all := Arr_1.all;
32
33  Show_Array ("Arr_2", Arr_2);
34
35  -- Access value assignment
36  Arr_2 := Arr_1;
37
38  Arr_1.all := (377, 610, 987, 1597, 2584, 4181);
39
40  Show_Array ("Arr_2", Arr_2);
41 end Show_Array_Assignments;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_Assignments
MD5: 9b1f99af081000c28a6bf9b033127ea3

Runtime output

```

Arr_2 1 2 3 5 8 13
Arr_2 377 610 987 1597 2584 4181

```

Here, `Arr_2.all := Arr_1.all` is an array assignment, while `Arr_2 := Arr_1` is an access value assignment. By forcing the usage of the `.all` suffix, the distinction is clear. Implicit dereferencing, however, could be confusing here. (For example, the `.all` suffix in `Arr_2 := Arr_1.all` is an oversight by the programmer when the intention actually was to use access values on both sides.) Therefore, implicit dereferencing is only supported in those cases where there's no risk of ambiguities or oversights.

Records

Let's see an example of implicit dereferencing of a record:

Listing 1337: show_dereferencing.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Dereferencing is
4
5      type Rec is record
6          I : Integer;
7          F : Float;
8      end record;
9
10     type Rec_Access is access Rec;
11
12     R : constant Rec_Access := new Rec;
13 begin
14     R.all := (I => 1, F => 5.0);
15
16     Put_Line ("R.I: "
17              & Integer'Image (R.I));
18     Put_Line ("R.F: "
19              & Float'Image (R.F));
20 end Show_Dereferencing;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.  
↳Record_Implicit_Dereferencing  
MD5: 9af72502d04f128785f77dcc829d5d48
```

Runtime output

```
R.I: 1  
R.F: 5.00000E+00
```

Again, we can replace `R.all.I` by `R.I`, as record components are implicitly dereferenced. Also, we could use implicit dereference when assigning to record components individually:

```
R.I := 1;  
R.F := 5.0;
```

However, we have to write `R.all` when assigning to the whole record `R`.

Attributes

Finally, let's see an example of implicit dereference when using attributes:

Listing 1338: show_dereferencing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Show_Dereferencing is  
4  
5   type Integer_Array is  
6     array (Positive range <>) of Integer;  
7  
8   type Integer_Array_Access is  
9     access Integer_Array;  
10  
11   Arr : constant Integer_Array_Access :=  
12         new Integer_Array (1 .. 6);  
13 begin  
14   Put_Line  
15     ("Arr'First: "  
16       & Integer'Image (Arr'First));  
17   Put_Line  
18     ("Arr'Last: "  
19       & Integer'Image (Arr'Last));  
20  
21   Put_Line  
22     ("Arr'Component_Size: "  
23       & Integer'Image (Arr'Component_Size));  
24   Put_Line  
25     ("Arr.all'Component_Size: "  
26       & Integer'Image (Arr.all'Component_Size));  
27  
28   Put_Line  
29     ("Arr'Size: "  
30       & Integer'Image (Arr'Size));  
31   Put_Line  
32     ("Arr.all'Size: "  
33       & Integer'Image (Arr.all'Size));  
34 end Show_Dereferencing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Dereferencing.Array_Explicit_Dereferencing
 MD5: 5730e18c8d2ed5e26a4d7d325a46a7e9

Runtime output

```
Arr'First: 1
Arr'Last: 6
Arr'Component_Size: 32
Arr.all'Component_Size: 32
Arr'Size: 128
Arr.all'Size: 192
```

Here, we can write `Arr'First` and `Arr'Last` instead of `Arr.all'First` and `Arr.all'Last`, respectively, because `Arr` is implicitly dereferenced. The same applies to `Arr'Component_Size`. Note that we can write both `Arr'Size` and `Arr.all'Size`, but they have different meanings:

- `Arr'Size` is the size of the access object; while
- `Arr.all'Size` indicates the size of the actual array `Arr`.

In other words, the `Size` attribute is *not* implicitly dereferenced. In fact, any attribute that could potentially be ambiguous is not implicitly dereferenced. Therefore, in those cases, we must explicitly indicate (by using `.all` or not) how we want to use the attribute.

Summary

The following table summarizes all instances where implicit dereferencing is supported:

Entities	Standard Usage	Implicit Dereference
Array components	<code>Arr.all (I)</code>	<code>Arr (I)</code>
Array slices	<code>Arr.all (F .. L)</code>	<code>Arr (F .. L)</code>
Record components	<code>Rec.all.C</code>	<code>Rec.C</code>
Array attributes	<code>Arr.all'First</code>	<code>Arr'First</code>
	<code>Arr.all'First (N)</code>	<code>Arr'First (N)</code>
	<code>Arr.all'Last</code>	<code>Arr'Last</code>
	<code>Arr.all'Last (N)</code>	<code>Arr'Last (N)</code>
	<code>Arr.all'Range</code>	<code>Arr'Range</code>
	<code>Arr.all'Range (N)</code>	<code>Arr'Range (N)</code>
	<code>Arr.all'Length</code>	<code>Arr'Length</code>
	<code>Arr.all'Length (N)</code>	<code>Arr'Length (N)</code>
	<code>Arr.all'Component_Size</code>	<code>Arr'Component_Size</code>
	<code>T.all'Identity</code>	<code>T'Identity</code>
Task attributes	<code>T.all'Storage_Size</code>	<code>T'Storage_Size</code>
	<code>T.all'Terminated</code>	<code>T'Terminated</code>
	<code>T.all'Callable</code>	<code>T'Callable</code>
	<code>X.all'Tag</code>	<code>X'Tag</code>
Tagged type attributes	<code>X.all'Valid</code>	<code>X'Valid</code>
Other attributes	<code>X.all'Old</code>	<code>X'Old</code>
	<code>A.all'Constrained</code>	<code>A'Constrained</code>

In the Ada Reference Manual

- [4.1 Names](#)²⁴⁸
- [4.1.1 Indexed Components](#)²⁴⁹

- 4.1.2 Slices²⁵⁰
- 4.1.3 Selected Components²⁵¹
- 4.1.4 Attributes²⁵²

28.1.8 Ragged arrays

Ragged arrays — also known as jagged arrays — are non-uniform, multidimensional arrays. They can be useful to implement tables with varying number of coefficients, as we discuss as an example in this section.

Uniform multidimensional arrays

Consider an algorithm that processes data based on coefficients that depends on a selected quality level:

Quality level	Number of coefficients	#1	#2	#3	#4	#5
Simplified	1	0.15				
Better	3	0.02	0.16	0.27		
Best	5	0.01	0.08	0.12	0.20	0.34

(Note that this is just a bogus table with no real purpose, as we're not trying to implement any actual algorithm.)

We can implement this table as a two-dimensional array (Calc_Table), where each quality level has an associated array:

Listing 1339: data_processing.ads

```

1 package Data_Processing is
2
3   type Quality_Level is
4     (Simplified, Better, Best);
5
6 private
7
8   Calc_Table : constant array
9     (Quality_Level, 1 .. 5) of Float :=
10     (Simplified =>
11       (0.15, 0.00, 0.00, 0.00, 0.00),
12       Better   =>
13         (0.02, 0.16, 0.27, 0.00, 0.00),
14       Best     =>
15         (0.01, 0.08, 0.12, 0.20, 0.34));
16
17   Last : constant array
18     (Quality_Level) of Positive :=
19     (Simplified => 1,
20      Better     => 3,
21      Best       => 5);
22
23 end Data_Processing;
```

²⁴⁸ <http://www.ada-auth.org/standards/22rm/html/RM-4-1.html>

²⁴⁹ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-1.html>

²⁵⁰ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-2.html>

²⁵¹ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-3.html>

²⁵² <http://www.ada-auth.org/standards/22rm/html/RM-4-1-4.html>

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
 ↪Uniform_Table
 MD5: befa8d2b684ee20495f2dd6907dc44d4

Note that, in this implementation, we have a separate table Last that indicates the actual number of coefficients of each quality level.

Alternatively, we could use a record (Table_Coefficient) that stores the number of coefficients and the actual coefficients:

Listing 1340: data_processing.ads

```

1 package Data_Processing is
2
3     type Quality_Level is
4       (Simplified, Better, Best);
5
6     type Data is
7       array (Positive range <>) of Float;
8
9 private
10
11     type Table_Coefficient is record
12       Last : Positive;
13       Coef : Data (1 .. 5);
14     end record;
15
16     Calc_Table : constant array
17       (Quality_Level) of Table_Coefficient :=
18       (Simplified =>
19         (1, (0.15, 0.00, 0.00, 0.00, 0.00)),
20        Better    =>
21         (3, (0.02, 0.16, 0.27, 0.00, 0.00)),
22        Best      =>
23         (5, (0.01, 0.08, 0.12, 0.20, 0.34)));
24
25 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
 ↪Uniform_Table
 MD5: 4c8602f6ecede0ac1231838c0a0a54b7

In this case, we have a unidimensional array where each component (of Table_Coefficient type) contains an array (Coef) with the coefficients.

This is an example of a Process procedure that references the Calc_Table:

Listing 1341: data_processing-operations.ads

```

1 package Data_Processing.Operations is
2
3     procedure Process (D : in out Data;
4                       Q :      Quality_Level);
5
6 end Data_Processing.Operations;
```

Listing 1342: data_processing-operations.adb

```

1 package body Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level) is
5   begin
6     for I in D'Range loop
7       for J in 1 .. Calc_Table (Q).Last loop
8         -- ... * Calc_Table (Q).Coef (J)
9         null;
10        end loop;
11        -- D (I) := ...
12        null;
13      end loop;
14    end Process;
15
16 end Data_Processing.Operations;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
 ↪Uniform_Table
 MD5: 2b0d2cee265509e64e507cfa6289bdcc

Note that, to loop over the coefficients, we're using **for J in 1 .. Calc_Table (Q).Last loop** instead of **for J in Calc_Table (Q)'Range loop**. As we're trying to make a non-uniform array fit in a uniform array, we cannot simply loop over all elements using the **Range** attribute, but must be careful to use the correct number of elements in the loop instead.

Also, note that Calc_Table has 15 coefficients in total. Out of those coefficients, 6 coefficients (or 40 percent of the table) aren't being used. Naturally, this is wasted memory space. We can improve this by using ragged arrays.

Non-uniform multidimensional array

Ragged arrays are declared by using an access type to an array. By doing that, each array can be declared with a different size, thereby creating a non-uniform multidimensional array.

For example, we can declare a constant array Table as a ragged array:

Listing 1343: data_processing.ads

```

1 package Data_Processing is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   private
7
8   type Integer_Array_Access is
9     access constant Integer_Array;
10
11   Table : constant array (1 .. 3) of
12     Integer_Array_Access :=
13     (1 => new Integer_Array'(1 => 15),
14      2 => new Integer_Array'(1 => 12,
15                             2 => 15,
16                             3 => 20),
```

(continues on next page)

(continued from previous page)

```

17      3 => new Integer_Array'(1 => 12,
18                             2 => 15,
19                             3 => 20,
20                             4 => 20,
21                             5 => 25,
22                             6 => 30));
23
24 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
 ↪ Simple_Ragged_Array
 MD5: 28e044a43bf45585a0268c60d63c629e

Here, each component of Table is an access to another array. As each array is allocated via **new**, those arrays may have different sizes.

We can rewrite the example from the previous subsection using a ragged array for the Calc_Table:

Listing 1344: data_processing.ads

```

1 package Data_Processing is
2
3   type Quality_Level is
4     (Simplified, Better, Best);
5
6   type Data is
7     array (Positive range <>) of Float;
8
9 private
10
11   type Coefficients is access constant Data;
12
13   Calc_Table : constant array (Quality_Level) of
14     Coefficients :=
15     (Simplified =>
16      new Data'(1 => 0.15),
17      Better    =>
18      new Data'(0.02, 0.16, 0.27),
19      Best      =>
20      new Data'(0.01, 0.08, 0.12,
21               0.20, 0.34));
22
23 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Ragged_Arrays.
 ↪ Ragged_Table
 MD5: 0781b27cba27dbd1e74da54e425a1f4b

Now, we aren't wasting memory space because each data component has the right size that is required for each quality level. Also, we don't need to store the number of coefficients, as this information is automatically available from the array initialization — via the allocation of the Data array for the Coefficients type.

Note that the Coefficients type is defined as **access constant**. We discuss *access-to-constant types* (page 925) in more details later on.

This is the adapted Process procedure:

Listing 1345: data_processing-operations.ads

```

1 package Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level);
5
6 end Data_Processing.Operations;
```

Listing 1346: data_processing-operations.adb

```

1 package body Data_Processing.Operations is
2
3   procedure Process (D : in out Data;
4                     Q :           Quality_Level) is
5   begin
6     for I in D'Range loop
7       for J in Calc_Table (Q)'Range loop
8         -- ... * Calc_Table (Q).Coef (J)
9         null;
10      end loop;
11      -- D (I) := ...
12      null;
13    end loop;
14  end Process;
15
16 end Data_Processing.Operations;
```

Now, we can simply loop over the coefficients by writing `for J in Calc_Table (Q)'Range loop`, as each element of `Calc_Table` automatically has the correct range.

28.1.9 Aliasing

The term [aliasing](https://en.wikipedia.org/wiki/Aliasing_(computing))²⁵³ refers to objects in memory that we can access using more than a single reference. In Ada, if we allocate an object via `new`, we have a potentially aliased object. We can then have multiple references to this object:

Listing 1347: show_aliasing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliasing is
4   type Integer_Access is access Integer;
5
6   A1, A2 : Integer_Access;
7 begin
8   A1 := new Integer;
9   A2 := A1;
10
11   A1.all := 22;
12   Put_Line ("A1: " & Integer'Image (A1.all));
13   Put_Line ("A2: " & Integer'Image (A2.all));
14
15   A2.all := 24;
16   Put_Line ("A1: " & Integer'Image (A1.all));
17   Put_Line ("A2: " & Integer'Image (A2.all));
18 end Show_Aliasing;
```

Code block metadata

²⁵³ [https://en.wikipedia.org/wiki/Aliasing_\(computing\)](https://en.wikipedia.org/wiki/Aliasing_(computing))

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliasing_Via_Access
 MD5: 2fde6073cec9823a1a9d93aec82384e1

Runtime output

```
A1: 22
A2: 22
A1: 24
A2: 24
```

In this example, we access the object allocated via **new** by using either A1 or A2, as both refer to the same *aliased* object. In other words, A1 or A2 allow us to access the same object in memory.

Important

Note that aliasing is unrelated to renaming. For example, we could use renaming to write a program that looks similar to the one above:

Listing 1348: show_renaming.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Renaming is
4      A1 : Integer;
5      A2 : Integer renames A1;
6  begin
7      A1 := 22;
8      Put_Line ("A1: " & Integer'Image (A1));
9      Put_Line ("A2: " & Integer'Image (A2));
10
11     A2 := 24;
12     Put_Line ("A1: " & Integer'Image (A1));
13     Put_Line ("A2: " & Integer'Image (A2));
14 end Show_Renaming;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Renaming
 MD5: 99a47d02000b91f7464dffe994fd8ee6

Runtime output

```
A1: 22
A2: 22
A1: 24
A2: 24
```

Here, A1 or A2 are two different names for the same object. However, the object itself isn't aliased.

In the Ada Reference Manual

- [3.10 Access Types](#)²⁵⁴

²⁵⁴ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Aliased objects

As we discussed *previously* (page 883), we use **new** to create aliased objects on the heap. We can also use general access types to access objects that were created on the stack.

By default, objects created on the stack aren't aliased. Therefore, we have to indicate that an object is aliased by using the **aliased** keyword in the object's declaration: `Obj : aliased Integer;`.

Let's see an example:

Listing 1349: show_aliased_obj.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Obj is
4   type Integer_Access is access all Integer;
5
6   I_Var : aliased Integer;
7   A1    : Integer_Access;
8 begin
9   A1 := I_Var'Access;
10
11   A1.all := 22;
12   Put_Line ("A1: " & Integer'Image (A1.all));
13 end Show_Aliased_Obj;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_
↳ Aliased_Obj
MD5: 98c8e47d7c2b5df8075918b239a8d476

Runtime output

```
A1: 22
```

Here, we declare `I_Var` as an aliased integer variable and get a reference to it, which we assign to `A1`. Naturally, we could also have two accesses `A1` and `A2`:

Listing 1350: show_aliased_obj.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Obj is
4   type Integer_Access is access all Integer;
5
6   I_Var : aliased Integer;
7   A1, A2 : Integer_Access;
8 begin
9   A1 := I_Var'Access;
10   A2 := A1;
11
12   A1.all := 22;
13   Put_Line ("A1: " & Integer'Image (A1.all));
14   Put_Line ("A2: " & Integer'Image (A2.all));
15
16   A2.all := 24;
17   Put_Line ("A1: " & Integer'Image (A1.all));
18   Put_Line ("A2: " & Integer'Image (A2.all));
19
20 end Show_Aliased_Obj;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_
 ↪Aliased_Obj
 MD5: ac331285456462f05abe7e1fd5e3ca2b

Runtime output

```
A1: 22
A2: 22
A1: 24
A2: 24
```

In this example, both A1 and A2 refer to the I_Var variable.

Note that these examples make use of these two features:

1. The declaration of a general access type (Integer_Access) using **access all**.
2. The retrieval of a reference to I_Var using the **Access** attribute.

In the next sections, we discuss these features in more details.

In the Ada Reference Manual

- 3.3.1 Object Declarations²⁵⁵
- 3.10 Access Types²⁵⁶

General access modifiers

Let's now discuss how to declare general access types. In addition to the *standard* (pool-specific) access type declarations, Ada provides two access modifiers:

Type	Declaration
Access-to-variable	type T_Acc is access all T
Access-to-constant	type T_Acc is access constant T

Let's look at an example:

Listing 1351: integer_access_types.ads

```
1 package Integer_Access_Types is
2
3     type Integer_Access is
4       access Integer;
5
6     type Integer_Access_All is
7       access all Integer;
8
9     type Integer_Access_Const is
10      access constant Integer;
11
12 end Integer_Access_Types;
```

Code block metadata

²⁵⁵ <http://www.ada-auth.org/standards/22rm/html/RM-3-3-1.html>

²⁵⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Show_Access_Modifiers
MD5: 98ccaa703194ae88222ccc5a4400e967

As we've seen previously, we can use a type such as `Integer_Access` to allocate objects dynamically. However, we cannot use this type to refer to declared objects, for example. In this case, we have to use an access-to-variable type such as `Integer_Access_All`. Also, if we want to access constants — or access objects that we want to treat as constants —, we use a type such as `Integer_Access_Const`.

Access attribute

To get access to a variable or a constant, we make use of the **Access** attribute. For example, `I_Var'Access` gives us access to the `I_Var` object.

Let's look at an example of how to use the integer access types from the previous code snippet:

Listing 1352: integer_access_types.ads

```
1 package Integer_Access_Types is
2
3     type Integer_Access is
4         access Integer;
5
6     type Integer_Access_All is
7         access all Integer;
8
9     type Integer_Access_Const is
10        access constant Integer;
11
12     procedure Show;
13
14 end Integer_Access_Types;
```

Listing 1353: integer_access_types.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2
3 package body Integer_Access_Types is
4
5     I_Var : aliased Integer := 0;
6     Fact  : aliased constant Integer := 42;
7
8     Dyn_Ptr      : constant Integer_Access
9                   := new Integer'(30);
10    I_Var_Ptr     : constant Integer_Access_All
11                  := I_Var'Access;
12    I_Var_C_Ptr   : constant Integer_Access_Const
13                  := I_Var'Access;
14    Fact_Ptr      : constant Integer_Access_Const
15                  := Fact'Access;
16
17    procedure Show is
18    begin
19        Put_Line ("Dyn_Ptr: "
20                  & Integer'Image (Dyn_Ptr.all));
21        Put_Line ("I_Var_Ptr: "
22                  & Integer'Image (I_Var_Ptr.all));
23        Put_Line ("I_Var_C_Ptr: "
24                  & Integer'Image
```

(continues on next page)

(continued from previous page)

```

25         (I_Var_C_Ptr.all));
26     Put_Line ("Fact_Ptr: "
27             & Integer'Image (Fact_Ptr.all));
28     end Show;
29
30 end Integer_Access_Types;

```

Listing 1354: show_access_modifiers.adb

```

1  with Integer_Access_Types;
2
3  procedure Show_Access_Modifiers is
4  begin
5      Integer_Access_Types.Show;
6  end Show_Access_Modifiers;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Show_Access_Modifiers
 MD5: c9036f060859207ea14354b26dc8b981

Runtime output

```

Dyn_Ptr:      30
I_Var_Ptr:    0
I_Var_C_Ptr:  0
Fact_Ptr:     42

```

In this example, Dyn_Ptr refers to a dynamically allocated object, I_Var_Ptr refers to the I_Var variable, and Fact_Ptr refers to the Fact constant. We get access to the variable and the constant objects by using the **Access** attribute.

Also, we declare I_Var_C_Ptr as an access-to-constant, but we get access to the I_Var variable. This simply means the object I_Var_C_Ptr refers to is treated as a constant. Therefore, we can write I_Var := 22;, but we cannot write I_Var_C_Ptr.all := 22;.

** In the Ada Reference Manual**

- 3.10.2 Operations of Access Types²⁵⁷

Non-aliased objects

As mentioned earlier, by default, declared objects — which are allocated on the stack — aren't aliased. Therefore, we cannot get a reference to those objects. For example:

Listing 1355: show_access_error.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Access_Error is
4  type Integer_Access is access all Integer;
5      I_Var : Integer;
6      A1   : Integer_Access;
7  begin
8      A1 := I_Var'Access;

```

(continues on next page)

²⁵⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

(continued from previous page)

```

9
10     A1.all := 22;
11     Put_Line ("A1: " & Integer'Image (A1.all));
12 end Show_Access_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Access_Non_Aliased_Obj
MD5: 2a9904062eea96ae6dc209493d6f20d4

Build output

```
show_access_error.adb:8:10: error: prefix of "Access" attribute must be aliased
gprbuild: *** compilation phase failed
```

In this example, the compiler complains that we cannot get a reference to `I_Var` because `I_Var` is not aliased.

Ragged arrays using aliased objects

We can use aliased objects to declare *ragged arrays* (page 918). For example, we can rewrite a previous program using aliased constant objects:

Listing 1356: data_processing.ads

```

1 package Data_Processing is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6 private
7
8     type Integer_Array_Access is
9         access constant Integer_Array;
10
11     Tab_1 : aliased constant Integer_Array
12             := (1 => 15);
13     Tab_2 : aliased constant Integer_Array
14             := (12, 15, 20);
15     Tab_3 : aliased constant Integer_Array
16             := (12, 15, 20,
17                 20, 25, 30);
18
19     Table : constant array (1 .. 3) of
20             Integer_Array_Access :=
21         (1 => Tab_1'Access,
22          2 => Tab_2'Access,
23          3 => Tab_3'Access);
24
25 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Ragged_Array_Aliased_Objs
MD5: 7e284560c447c02628e34bac982d4ad5

Here, instead of allocating the constant arrays dynamically via `new`, we declare three aliased arrays (`Tab_1`, `Tab_2` and `Tab_3`) and get a reference to them in the declaration of `Table`.

Aliased access objects

It's interesting to mention that access objects can be aliased themselves. Consider this example where we declare the `Integer_Access_Access` type to refer to an access object:

Listing 1357: show_aliased_access_obj.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Access_Obj is
4
5     type Integer_Access is
6       access all Integer;
7     type Integer_Access_Access is
8       access all Integer_Access;
9
10    I_Var : aliased Integer;
11    A      : aliased Integer_Access;
12    B      : Integer_Access_Access;
13 begin
14     A := I_Var'Access;
15     B := A'Access;
16
17     B.all.all := 22;
18     Put_Line ("A: " & Integer'Image (A.all));
19     Put_Line ("B: " & Integer'Image (B.all.all));
20 end Show_Aliased_Access_Obj;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_Access
 MD5: 77e9be5e29cfb99aef9409728202ba9d

Runtime output

```

A: 22
B: 22
```

After the assignments in this example, B refers to A, which in turn refers to I_Var. Note that this code only compiles because we declare A as an aliased (access) object.

Aliased components

Components of an array or a record can be aliased. This allows us to get access to those components:

Listing 1358: show_aliased_components.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Aliased_Components is
4
5     type Integer_Access is access all Integer;
6
7     type Rec is record
8       I_Var_1 : Integer;
9       I_Var_2 : aliased Integer;
10    end record;
11
12    type Integer_Array is
13      array (Positive range <>) of aliased Integer;
```

(continues on next page)

(continued from previous page)

```

14
15   R   : Rec := (22, 24);
16   Arr : Integer_Array (1 .. 3) := (others => 42);
17   A   : Integer_Access;
18 begin
19   -- A := R.I_Var_1'Access;
20   --      ^ ERROR: cannot access
21   --      non-aliased
22   --      component
23
24   A := R.I_Var_2'Access;
25   Put_Line ("A: " & Integer'Image (A.all));
26
27   A := Arr (2)'Access;
28   Put_Line ("A: " & Integer'Image (A.all));
29 end Show_Aliased_Components;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_Components
MD5: 5dfaa248caf8e37a4a3a1e1a24973777

Runtime output

```

A:  24
A:  42

```

In this example, we get access to the `I_Var_2` component of record `R`. (Note that trying to access the `I_Var_1` component would give us a compilation error, as this component is not aliased.) Similarly, we get access to the second component of array `Arr`.

Declaring components with the **aliased** keyword allows us to specify that those are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This can be essential when doing low-level programming — for example, when accessing memory-mapped registers. In this case, we want to ensure that the compiler uses the memory address we're specifying (instead of assigning registers for those components).

In the Ada Reference Manual

- [3.6 Array Types](#)²⁵⁸

Aliased parameters

In addition to aliased objects and components, we can declare *aliased parameters* (page 769), as we already discussed in an earlier chapter. As we mentioned there, aliased parameters are always passed by reference, independently of the type we're using.

The parameter mode indicates which type we must use for the access type:

Parameter mode	Type
aliased in	Access-to-constant
aliased out	Access-to-variable
aliased in out	Access-to-variable

²⁵⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-6.html>

Using aliased parameters in a subprogram allows us to get access to those parameters in the body of that subprogram. Let's see an example:

Listing 1359: data_processing.ads

```

1 package Data_Processing is
2
3     procedure Proc (I : aliased in out Integer);
4
5 end Data_Processing;
```

Listing 1360: data_processing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Data_Processing is
4
5     procedure Show (I : aliased Integer) is
6         --      ^ equivalent to
7         --      "aliased in Integer"
8
9     type Integer_Constant_Access is
10        access constant Integer;
11
12     A : constant Integer_Constant_Access
13        := I'Access;
14 begin
15     Put_Line ("Value : I "
16             & Integer'Image (A.all));
17 end Show;
18
19 procedure Set_One (I : aliased out Integer) is
20
21     type Integer_Access is access all Integer;
22
23     procedure Local_Set_One (A : Integer_Access)
24     is
25     begin
26         A.all := 1;
27     end Local_Set_One;
28
29 begin
30     Local_Set_One (I'Access);
31 end Set_One;
32
33 procedure Proc (I : aliased in out Integer) is
34
35     type Integer_Access is access all Integer;
36
37     procedure Add_One (A : Integer_Access) is
38     begin
39         A.all := A.all + 1;
40     end Add_One;
41
42 begin
43     Show (I);
44     Add_One (I'Access);
45     Show (I);
46 end Proc;
47
48 end Data_Processing;
```

Listing 1361: show_aliased_param.adb

```

1 with Data_Processing; use Data_Processing;
2
3 procedure Show_Aliased_Param is
4   I : aliased Integer := 22;
5 begin
6   Proc (I);
7 end Show_Aliased_Param;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Aliasing.Aliased_Rec_Component
 MD5: 076238603036aa51cafcc013f38bc8f3

Runtime output

```

Value : I  22
Value : I  23

```

Here, Proc has an **aliased in out** parameter. In Proc's body, we declare the Integer_Access type as an **access all** type. We use the same approach in body of the Set_One procedure, which has an **aliased out** parameter. Finally, the Show procedure has an **aliased in** parameter. Therefore, we declare the Integer_Constant_Access as an **access constant** type.

Note that parameter aliasing has an influence on how arguments are passed to a subprogram when the parameter is of scalar type. When a scalar parameter is declared as aliased, the corresponding argument is passed by reference. For example, if we had declared **procedure Show (I : Integer)**, the argument for I would be passed by value. However, since we're declaring it as **aliased Integer**, it is passed by reference.

i In the Ada Reference Manual

- 6.1 Subprogram Declarations²⁵⁹
- 6.2 Formal Parameter Modes²⁶⁰
- 6.4.1 Parameter Associations²⁶¹

28.1.10 Accessibility Levels and Rules: An Introduction

This section provides an introduction to accessibility levels and accessibility rules. This topic can be very complicated, and by no means do we intend to cover all the details here. (In fact, discussing all the details about accessibility levels and rules could be a long chapter on its own. If you're interested in them, please refer to the Ada Reference Manual.) In any case, the goal of this section is to present the intention behind the accessibility rules and build intuition on how to best use access types in your code.

i In the Ada Reference Manual

- 3.10.2 Operations of Access Types²⁶²

²⁵⁹ <http://www.ada-auth.org/standards/22rm/html/RM-6-1.html>

²⁶⁰ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

²⁶¹ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²⁶² <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

Lifetime of objects

First, let's talk a bit about [lifetime of objects](#)²⁶³. We assume you understand the concept, so this section is very short.

In very simple terms, the lifetime of an object indicates when an object still has relevant information. For example, if a variable *V* gets out of scope, we say that its lifetime has ended. From this moment on, *V* no longer exists.

For example:

Listing 1362: show_lifetime.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Lifetime is
4      I_Var_1 : Integer := 22;
5  begin
6
7      Inner_Block : declare
8          I_Var_2 : Integer := 42;
9      begin
10         Put_Line ("I_Var_1: "
11                 & Integer'Image (I_Var_1));
12         Put_Line ("I_Var_2: "
13                 & Integer'Image (I_Var_2));
14
15         -- I_Var_2 will get out of scope
16         -- when the block finishes.
17     end Inner_Block;
18
19     -- I_Var_2 is now out of scope...
20
21     Put_Line ("I_Var_1: "
22             & Integer'Image (I_Var_1));
23     Put_Line ("I_Var_2: "
24             & Integer'Image (I_Var_2));
25     --
26     -- ERROR: lifetime of I_Var_2 has ended!
27 end Show_Lifetime;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Lifetime
MD5: ebe36f12c832ecfe71399b89801808d4

Build output

```

show_lifetime.adb:24:31: error: "I_Var_2" is undefined
gprbuild: *** compilation phase failed
```

In this example, we declare *I_Var_1* in the *Show_Lifetime* procedure, and *I_Var_2* in its *Inner_Block*.

This example doesn't compile because we're trying to use *I_Var_2* after its lifetime has ended. However, if such a code could compile and run, the last call to *Put_Line* would potentially display garbage to the user. (In fact, the actual behavior would be undefined.)

²⁶³ [https://en.wikipedia.org/wiki/Variable_\(computer_science\)#Scope_and_extent](https://en.wikipedia.org/wiki/Variable_(computer_science)#Scope_and_extent)

Accessibility Levels

In basic terms, accessibility levels are a mechanism to assess the lifetime of objects (as we've just discussed). The starting point is the library level: this is the base level, and no level can be deeper than that. We start "moving" to deeper levels when we use a library in a subprogram or call other subprograms for example.

Suppose we have a procedure `Proc` that makes use of a package `Pkg`, and there's a block in the `Proc` procedure:

```
package Pkg is
    -- Library level
end Pkg;

with Pkg; use Pkg;

procedure Proc is
    -- One level deeper than
    -- library level
begin
    declare
        -- Two levels deeper than
        -- library level
    begin
        null;
    end;
end Proc;
```

For this code, we can say that:

- the specification of `Pkg` is at library level;
- the declarative part of `Proc` is one level deeper than the library level; and
- the block is two levels deeper than the library level.

(Note that this is still a very simplified overview of accessibility levels. Things start getting more complicated when we use information from `Pkg` in `Proc`. Those details will become more clear in the next sections.)

The levels themselves are not visible to the programmer. For example, there's no `Access_Level` attribute that returns an integer value indicating the level. Also, you cannot write a user message that displays the level at a certain point. In this sense, accessibility levels are assessed relatively to each other: we can only say that a specific operation is at the same or at a deeper level than another one.

Accessibility Rules

The accessibility rules determine whether a specific use of access types or objects is legal (or not). Actually, accessibility rules exist to prevent *dangling references* (page 939), which we discuss later. Also, they are based on the *accessibility levels* (page 934) we discussed earlier.

Code example

As mentioned earlier, the accessibility level at a specific point isn't visible to the programmer. However, to illustrate which level we have at each point in the following code example, we use a prefix (L0, L1, and L2) to indicate whether we're at the library level (L0) or at a deeper level.

Let's now look at the complete code example:

Listing 1363: library_level.ads

```

1 package Library_Level is
2
3     type L0_Integer_Access is
4       access all Integer;
5
6     L0_IA  : L0_Integer_Access;
7
8     L0_Var : aliased Integer;
9
10 end Library_Level;
```

Listing 1364: show_library_level.adb

```

1 with Library_Level; use Library_Level;
2
3 procedure Show_Library_Level is
4     type L1_Integer_Access is
5       access all Integer;
6
7     L0_IA_2 : L0_Integer_Access;
8     L1_IA   : L1_Integer_Access;
9
10    L1_Var : aliased Integer;
11
12    procedure Test is
13        type L2_Integer_Access is
14          access all Integer;
15
16        L2_IA  : L2_Integer_Access;
17
18        L2_Var : aliased Integer;
19    begin
20        L1_IA := L2_Var'Access;
21        --      ^^^^^^
22        --      ILLEGAL: L2 object to
23        --                  L1 access object
24
25        L2_IA := L2_Var'Access;
26        --      ^^^^^^
27        --      LEGAL: L2 object to
28        --                  L2 access object
29    end Test;
30
31 begin
32     L0_IA := new Integer'(22);
33     --      ^^^^^^^^^^
34     --      LEGAL: L0 object to
35     --                  L0 access object
36
37     L0_IA_2 := new Integer'(22);
38     --      ^^^^^^^^^^
```

(continues on next page)

(continued from previous page)

```

39      --      LEGAL: L0 object to
40      --      L0 access object
41
42      L0_IA := L1_Var'Access;
43      --      ^^^^^^
44      --      ILLEGAL: L1 object to
45      --      L0 access object
46
47      L0_IA_2 := L1_Var'Access;
48      --      ^^^^^^
49      --      ILLEGAL: L1 object to
50      --      L0 access object
51
52      L1_IA := L0_Var'Access;
53      --      ^^^^^^
54      --      LEGAL: L0 object to
55      --      L1 access object
56
57      L1_IA := L1_Var'Access;
58      --      ^^^^^^
59      --      LEGAL: L1 object to
60      --      L1 access object
61
62      L0_IA := L1_IA;
63      --      ^^^^^^
64      --      ILLEGAL: type mismatch
65
66      L0_IA := L0_Integer_Access (L1_IA);
67      --      ^^^^^^^^^^^^^^^^^^
68      --      ILLEGAL: cannot convert
69      --      L1 access object to
70      --      L0 access object
71
72      Test;
73  end Show_Library_Level;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Accessibility_Library_Level
 MD5: b3bed7eb2a8dfc78a2e7a7d2ce99f736

Build output

```

show_library_level.adb:20:16: error: non-local pointer cannot point to local object
show_library_level.adb:42:13: error: non-local pointer cannot point to local object
show_library_level.adb:47:15: error: non-local pointer cannot point to local object
show_library_level.adb:62:13: error: expected type "L0_Integer_Access" defined at
↳ library_level.ads:3
show_library_level.adb:62:13: error: found type "L1_Integer_Access" defined at
↳ line 4
show_library_level.adb:66:32: error: cannot convert local pointer to non-local
↳ access type
gprbuild: *** compilation phase failed

```

In this example, we declare

- in the `Library_Level` package: the `L0_Integer_Access` type, the `L0_IA` access object, and the `L0_Var` aliased variable;
- in the `Show_Library_Level` procedure: the `L1_Integer_Access` type, the `L0_IA_2` and `L1_IA` access objects, and the `L1_Var` aliased variable;

- in the nested Test procedure: the L2_Integer_Access type, the L2_IA, and the L2_Var aliased variable.

As mentioned earlier, the Ln prefix indicates the level of each type or object. Here, the n value is zero at library level. We then increment the n value each time we refer to a deeper level.

For instance:

- when we declare the L1_Integer_Access type in the Show_Library_Level procedure, that declaration is one level deeper than the level of the Library_Level package — so it has the L1 prefix.
- when we declare the L2_Integer_Access type in the Test procedure, that declaration is one level deeper than the level of the Show_Library_Level procedure — so it has the L2 prefix.

Types and Accessibility Levels

It's very important to highlight the fact that:

- types themselves also have an associated level, and
- objects have the same accessibility level as their types.

When we declare the L0_IA_2 object in the code example, its accessibility level is at library level because its type (the L0_Integer_Access type) is at library level. Even though this declaration is in the Show_Library_Level procedure — whose declarative part is one level deeper than the library level —, the object itself has the same accessibility level as its type.

Now that we've discussed the accessibility levels of this code example, let's see how the accessibility rules use those levels.

Operations on Access Types

In very simple terms, the accessibility rules say that:

- operations on access types at the same accessibility level are legal;
- assigning or converting to a deeper level is legal;

Otherwise, operations targeting objects at a *less-deep* level are illegal.

For example, `L0_IA := new Integer'(22)` and `L1_IA := L1_Var'Access` are legal because we're operating at the same accessibility level. Also, `L1_IA := L0_Var'Access` is legal because L1_IA is at a deeper level than L0_Var'Access.

However, many operations in the code example are illegal. For instance, `L0_IA := L1_Var'Access` and `L0_IA_2 := L1_Var'Access` are illegal because the target objects in the assignment are *less deep*.

Note that the `L0_IA := L1_IA` assignment is mainly illegal because the access types don't match. (Of course, in addition to that, assigning `L1_Var'Access` to `L0_IA` is also illegal in terms of accessibility rules.)

Conversion between Access Types

The same rules apply to the conversion between access types. In the code example, the `L0_Integer_Access (L1_IA)` conversion is illegal because the resulting object is less deep. That being said, conversions on the same level are fine:

Listing 1365: show_same_level_conversion.adb

```

1  procedure Show_Same_Level_Conversion is
2      type L1_Integer_Access is
3          access all Integer;
4
5      type L1_B_Integer_Access is
6          access all Integer;
7
8      L1_IA    : L1_Integer_Access;
9      L1_B_IA : L1_B_Integer_Access;
10
11     L1_Var   : aliased Integer;
12 begin
13     L1_IA := L1_Var'Access;
14
15     L1_B_IA := L1_B_Integer_Access (L1_IA);
16     --      ~~~~~
17     --      LEGAL: conversion from
18     --              L1 access object to
19     --              L1 access object
20 end Show_Same_Level_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Same_Level_Conversion
MD5: 7276a06e9f5b634d4f5a10a892071d87

Here, we're converting from the L1_Integer_Access type to the L1_B_Integer_Access, which are both at the same level.

Accessibility rules on parameters

Note that the accessibility rules also apply to access values as subprogram parameters. For example, compilation fails for this example:

Listing 1366: names.ads

```

1  package Names is
2
3      type Name is access all String;
4
5      type Constant_Name is
6          access constant String;
7
8      procedure Show (N : Constant_Name);
9
10 end Names;

```

Listing 1367: names.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  -- with Ada.Characters.Handling;
4  -- use Ada.Characters.Handling;
5
6  package body Names is
7
8      procedure Show (N : Constant_Name) is
9      begin

```

(continues on next page)

(continued from previous page)

```

10     -- for I in N'Range loop
11     --     N (I) := To_Lower (N (I));
12     -- end loop;
13     Put_Line ("Name: " & N.all);
14 end Show;
15
16 end Names;

```

Listing 1368: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     S : aliased String := "John";
5 begin
6     Show (S'Access);
7 end Show_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Accessibility_Checks_Parameters
 MD5: 6b8bf2799caa32f55d216ac0b58fcd39

Build output

```

show_names.adb:6:10: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed

```

In this case, the `S'Access` cannot be used as the actual parameter for the `N` parameter of the `Show` procedure because it's in a deeper level. If we allocate the string via `new`, however, the code compiles as expected:

Listing 1369: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     S : Name := new String ("John");
5 begin
6     Show (Constant_Name (S));
7 end Show_Names;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Accessibility_Checks_Parameters
 MD5: 30237c83426db758804b802e1953d5d9

Runtime output

```
Name: John
```

This version of the code works because both object and access object have the same level.

Dangling References

An access value that points to a non-existent object is called a dangling reference. Later on, we'll discuss how dangling references may occur using *unchecked deallocation* (page 947).

Dangling references are created when we have an access value pointing to an object whose lifetime has ended, so it becomes a non-existent object. This could occur, for example, when an access value still points to an object X that has gone out of scope.

As mentioned in the previous section, the accessibility rules of the Ada language ensure that such situations never happen! In fact, whenever possible, the compiler applies those rules to detect potential dangling references at compile time. When this detection isn't possible at compile time, the compiler introduces an *accessibility check* (page 815). If this check fails at runtime, it raises a Program_Error exception — thereby preventing that a dangling reference gets used.

Let's see an example of how dangling references could occur:

Listing 1370: show_dangling_reference.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Dangling_Reference is
4
5      type Integer_Access is
6          access all Integer;
7
8      I_Var_1 : aliased Integer := 22;
9
10     A1      : Integer_Access;
11  begin
12     A1 := I_Var_1'Access;
13     Put_Line ("A1.all: "
14              & Integer'Image (A1.all));
15
16     Put_Line ("Inner_Block will start now!");
17
18     Inner_Block : declare
19         --
20         -- I_Var_2 only exists in Inner_Block
21         --
22         I_Var_2 : aliased Integer := 42;
23
24         --
25         -- A2 only exists in Inner_Block
26         --
27         A2      : Integer_Access;
28     begin
29         A2 := I_Var_1'Access;
30         Put_Line ("A2.all: "
31                  & Integer'Image (A2.all));
32
33         A1 := I_Var_2'Access;
34         -- PROBLEM: A1 and Integer_Access type
35         --           have longer lifetime than
36         --           I_Var_2
37
38         Put_Line ("A1.all: "
39                  & Integer'Image (A1.all));
40
41         A2 := I_Var_2'Access;
42         -- PROBLEM: A2 has the same lifetime as
43         --           I_Var_2, but Integer_Access
44         --           type has a longer lifetime.
45
46         Put_Line ("A2.all: "
47                  & Integer'Image (A2.all));
48     end Inner_Block;

```

(continues on next page)

(continued from previous page)

```

49
50   Put_Line ("Inner_Block has ended!");
51   Put_Line ("A1.all: "
52             & Integer'Image (A1.all));
53
54 end Show_Dangling_Reference;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
 ↳ Levels_Rules_Introduction.Dangling_Reference_Rules
 MD5: 98e597f3f6a12075c474612bb42f4cb7

Build output

```

show_dangling_reference.adb:33:13: error: non-local pointer cannot point to local_
↳ object
show_dangling_reference.adb:41:13: error: non-local pointer cannot point to local_
↳ object
gprbuild: *** compilation phase failed
```

Here, we declare the access objects A1 and A2 of Integer_Access type, and the I_Var_1 and I_Var_2 objects. Moreover, A1 and I_Var_1 are declared in the scope of the Show_Dangling_Reference procedure, while A2 and I_Var_2 are declared in the Inner_Block.

When we try to compile this code, we get two compilation errors due to violation of accessibility rules. Let's now discuss these accessibility rules in terms of lifetime, and see which problems they are preventing in each case.

1. In the `A1 := I_Var_2'Access` assignment, the main problem is that A1 has a longer lifetime than I_Var_2. After the Inner_Block finishes — when I_Var_2 gets out of scope and its lifetime has ended —, A1 would still be pointing to an object that does not longer exist.
2. In the `A2 := I_Var_2'Access` assignment, however, both A2 and I_Var_2 have the same lifetime. In that sense, the assignment may actually look pretty much OK.
 - However, as mentioned in the previous section, Ada also cares about the lifetime of access types. In fact, since the Integer_Access type is declared outside of the Inner_Block, it has a longer lifetime than A2 and I_Var_2.
 - To be more precise, the accessibility rules detect that A2 is an access object of a type that has a longer lifetime than I_Var_2.

At first glance, this last accessibility rule may seem too strict, as both A2 and I_Var_2 have the same lifetime — so nothing bad could occur when dereferencing A2. However, consider the following change to the code:

```

A2 := I_Var_2'Access;

A1 := A2;
--   PROBLEM: A1 will still be referring
--             to I_Var_2 after the
--             Inner_Block, i.e. when the
--             lifetime of I_Var_2 has
--             ended!
```

Here, we're introducing the `A1 := A2` assignment. The problem with this is that I_Var_2's lifetime ends when the Inner_Block finishes, but A1 would continue to refer to an I_Var_2 object that doesn't exist anymore — thereby creating a dangling reference.

Even though we're actually not assigning A2 to A1 in the original code, we could have done it. The accessibility rules ensure that such an error is never introduced into the program.

i For further reading...

In the original code, we can consider the `A2 := I_Var_2'Access` assignment to be safe, as we're not using the `A1 := A2` assignment there. Since we're confident that no error could ever occur in the `Inner_Block` due to the assignment to A2, we could replace it with `A2 := I_Var_2'Unchecked_Access`, so that the compiler accepts it. We discuss more about the unchecked access attribute *later in this chapter* (page 942).

Alternatively, we could have solved the compilation issue that we see in the `A2 := I_Var_2'Access` assignment by declaring another access type locally in the `Inner_Block`:

```
Inner_Block : declare
  type Integer_Local_Access is
    access all Integer;

  I_Var_2 : aliased Integer := 42;

  A2      : Integer_Local_Access;
begin
  A2 := I_Var_2'Access;
  -- This assignment is fine because
  -- the Integer_Local_Access type has
  -- the same lifetime as I_Var_2.
end Inner_Block;
```

With this change, A2 becomes an access object of a type that has the same lifetime as `I_Var_2`, so that the assignment doesn't violate the rules anymore.

(Note that in the `Inner_Block`, we could have simply named the local access type `Integer_Access` instead of `Integer_Local_Access`, thereby masking the `Integer_Access` type of the outer block.)

We discuss the effects of dereferencing dangling references *later in this chapter* (page 949).

28.1.11 Unchecked Access

In this section, we discuss the `Unchecked_Access` attribute, which we can use to circumvent accessibility issues for objects in specific cases. (Note that this attribute only exists for objects, not for subprograms.)

We've seen *previously* (page 932) that the accessibility levels verify the lifetime of access types. Let's see a simplified version of a code example from that section:

Listing 1371: integers.ads

```
1 package Integers is
2
3   type Integer_Access is access all Integer;
4
5 end Integers;
```

Listing 1372: show_access_issue.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Integers; use Integers;
```

(continues on next page)

(continued from previous page)

```

4
5 procedure Show_Access_Issue is
6   I_Var : aliased Integer := 42;
7
8   A      : Integer_Access;
9 begin
10  A := I_Var'Access;
11  -- PROBLEM: A has the same lifetime as I_Var,
12  --           but Integer_Access type has a
13  --           longer lifetime.
14
15  Put_Line ("A.all: " & Integer'Image (A.all));
16 end Show_Access_Issue;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Access.
 ↳Dangling_Reference_Rules
 MD5: 646acabf3f388b52809349463d20d314

Build output

```

show_access_issue.adb:10:09: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed

```

Here, the compiler complains about the `A := I_Var'Access` assignment because the `Integer_Access` type has a longer lifetime than `A`. However, we know that this assignment to `A` — and further uses of `A` in the code — won't cause dangling references to be created. Therefore, we can assume that assigning the access to `I_Var` to `A` is safe.

When we're sure that an access assignment cannot possibly generate dangling references, we can use the `Unchecked_Access` attribute. For instance, we can use this attribute to circumvent the compilation error in the previous code example, since we know that the assignment is actually safe:

Listing 1373: integers.ads

```

1 package Integers is
2
3   type Integer_Access is access all Integer;
4
5 end Integers;

```

Listing 1374: show_access_issue.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Integers; use Integers;
4
5 procedure Show_Access_Issue is
6   I_Var : aliased Integer := 42;
7
8   A      : Integer_Access;
9 begin
10  A := I_Var'Unchecked_Access;
11  -- OK: assignment is now accepted.
12
13  Put_Line ("A.all: " & Integer'Image (A.all));
14 end Show_Access_Issue;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Access.
↳Dangling_Reference_Rules
MD5: a71b9076d9e2983ffb9811183afdf6c1

Runtime output

A.all: 42

When we use the `Unchecked_Access` attribute, most rules still apply. The only difference to the standard `Access` attribute is that unchecked access applies the rules as if the object we're getting access to was being declared at library level. (For the code example we've just seen, the check would be performed as if `I_Var` was declared in the `Integers` package instead of being declared in the procedure.)

It is strongly recommended to avoid unchecked access in general. You should only use it when you can safely assume that the access object will be discarded before the object we had access to gets out of scope. Therefore, if this situation isn't clear enough, it's best to avoid unchecked access. (Later in this chapter, we'll see some of the nasty issues that arrive from creating dangling references.) Instead, you should work on improving the software design of your application by considering alternatives such as using containers or encapsulating access types in well-designed abstract data types.

In the Ada Reference Manual

- [Unchecked Access Value Creation](#)²⁶⁴

28.1.12 Unchecked Deallocation

So far, we've seen multiple examples of using `new` to allocate objects. In this section, we discuss how to manually deallocate objects.

Our starting point to manually deallocate an object is the generic `Ada.Unchecked_Deallocation` procedure. We first instantiate this procedure for an access type whose objects we want to be able to deallocate. For example, let's instantiate it for the `Integer_Access` type:

Listing 1375: `integer_types.ads`

```
1 with Ada.Unchecked_Deallocation;
2
3 package Integer_Types is
4
5     type Integer_Access is access Integer;
6
7     --
8     --  Instantiation of Ada.Unchecked_Deallocation
9     --  for the Integer_Access type:
10    --
11    procedure Free is
12        new Ada.Unchecked_Deallocation
13            (Object => Integer,
14             Name   => Integer_Access);
15 end Integer_Types;
```

Code block metadata

²⁶⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-10.html>

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Simple_Unchecked_Deallocation
 MD5: 328b244cf406853e87494c381c9c4c9e

Here, we declare the `Free` procedure, which we can then use to deallocate objects that were allocated for the `Integer_Access` type.

`Ada.Unchecked_Deallocation` is a generic procedure that we can instantiate for access types. When declaring an instance of `Ada.Unchecked_Deallocation`, we have to specify arguments for:

- the formal `Object` parameter, which indicates the type of actual objects that we want to deallocate; and
- the formal `Name` parameter, which indicates the access type.

In a type declaration such as `type Integer_Access is access Integer`, `Integer` denotes the `Object`, while `Integer_Access` denotes the `Name`.

Because each instance of `Ada.Unchecked_Deallocation` is bound to a specific access type, we cannot use it for another access type, even if the type we use for the `Object` parameter is the same:

Listing 1376: `integer_types.ads`

```

1 with Ada.Unchecked_Deallocation;
2
3 package Integer_Types is
4
5   type Integer_Access is access Integer;
6
7   procedure Free is
8     new Ada.Unchecked_Deallocation
9       (Object => Integer,
10        Name   => Integer_Access);
11
12   type Another_Integer_Access is access Integer;
13
14   procedure Free is
15     new Ada.Unchecked_Deallocation
16       (Object => Integer,
17        Name   => Another_Integer_Access);
18 end Integer_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Simple_Unchecked_Deallocation
 MD5: b9bc58ff60632287237e2e322fc6c63e

Here, we're declaring two `Free` procedures: one for the `Integer_Access` type, another for the `Another_Integer_Access`. We cannot use the `Free` procedure for the `Integer_Access` type when deallocating objects associated with the `Another_Integer_Access` type, even though both types are declared as `access Integer`.

Note that we can use any name when instantiating the `Ada.Unchecked_Deallocation` procedure. However, naming it `Free` is very common.

Now, let's see a complete example that includes object allocation and deallocation:

Listing 1377: integer_types.ads

```
1 with Ada.Unchecked_Deallocation;
2
3 package Integer_Types is
4
5     type Integer_Access is access Integer;
6
7     procedure Free is
8         new Ada.Unchecked_Deallocation
9             (Object => Integer,
10              Name   => Integer_Access);
11
12     procedure Show_Is_Null (I : Integer_Access);
13
14 end Integer_Types;
```

Listing 1378: integer_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Integer_Types is
4
5     procedure Show_Is_Null (I : Integer_Access) is
6     begin
7         if I = null then
8             Put_Line ("access value is null.");
9         else
10            Put_Line ("access value is NOT null.");
11        end if;
12    end Show_Is_Null;
13
14 end Integer_Types;
```

Listing 1379: show_unchecked_deallocation.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Integer_Types; use Integer_Types;
3
4 procedure Show_Unchecked_Deallocation is
5
6     I : Integer_Access;
7
8 begin
9     Put ("We haven't called new yet... ");
10    Show_Is_Null (I);
11
12    Put ("Calling new... ");
13    I := new Integer;
14    Show_Is_Null (I);
15
16    Put ("Calling Free... ");
17    Free (I);
18    Show_Is_Null (I);
19 end Show_Unchecked_Deallocation;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
↳ Deallocation.Unchecked_Deallocation
MD5: a9f2df04e2fe0d5ee8c17249b4ae315a

Runtime output

```
We haven't called new yet... access value is null.
Calling new... access value is NOT null.
Calling Free... access value is null.
```

In the `Show_Unchecked_Deallocation` procedure, we first allocate an object for `I` and then call `Free (I)` to deallocate it. Also, we call the `Show_Is_Null` procedure at three different points: before any allocation takes place, after allocating an object for `I`, and after deallocating that object.

When we deallocate an object via a call to `Free`, the corresponding access value — which was previously pointing to an existing object — is set to `null`. Therefore, `I = null` after the call to `Free`, which is exactly what we see when running this example code.

Note that it is OK to call `Free` multiple times for the same access object:

Listing 1380: `show_unchecked_deallocation.adb`

```
1 with Integer_Types; use Integer_Types;
2
3 procedure Show_Unchecked_Deallocation is
4
5     I : Integer_Access;
6
7 begin
8     I := new Integer;
9
10    Free (I);
11    Free (I);
12    Free (I);
13 end Show_Unchecked_Deallocation;
```

Code block metadata

Project: `Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Unchecked_Deallocation`
MD5: `ce7f4f912f12d723ca673ca36a478765`

The multiple calls to `Free` for the same access object don't cause any issues. Because the access value is `null` after the first call to `Free (I)`, we're actually just passing `null` as an argument in the second and third calls to `Free`. However, any attempt to deallocate an access value of `null` is ignored in the `Free` procedure, so the second and third calls to `Free` don't have any effect.

In the Ada Reference Manual

- [4.8 Allocators](#)²⁶⁵
- [13.11.2 Unchecked Storage Deallocation](#)²⁶⁶

Unchecked Deallocation and Dangling References

We've discussed *dangling references* (page 939) before. In this section, we discuss how unchecked deallocation can create dangling references and the issues of having them in an application.

Let's reuse the last example and introduce `I_2`, which will point to the same object as `I`:

²⁶⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-8.html>

²⁶⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html>

Listing 1381: show_unchecked_deallocation.adb

```

1  with Integer_Types; use Integer_Types;
2
3  procedure Show_Unchecked_Deallocation is
4
5      I, I_2 : Integer_Access;
6
7  begin
8      I := new Integer;
9
10     I_2 := I;
11
12     -- NOTE: I_2 points to the same
13     --       object as I.
14
15     --
16     -- Use I and I_2...
17     --
18     -- ... then deallocate memory...
19     --
20
21     Free (I);
22
23     -- NOTE: at this point, I_2 is a
24     --       dangling reference!
25
26     -- Further calls to Free (I)
27     -- are OK!
28
29     Free (I);
30     Free (I);
31
32     -- A call to Free (I_2) is
33     -- NOT OK:
34
35     Free (I_2);
36 end Show_Unchecked_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Unchecked_Deallocation
MD5: ee5c20209a113a6c1bc7895b8ebdb174

Runtime output

```

free(): double free detected in tcache 2

raised PROGRAM_ERROR : unhandled signal

```

As we've seen before, we can have multiple calls to `Free (I)`. However, the call to `Free (I_2)` is bad because `I_2` is not null. In fact, it is a dangling reference — i.e. `I_2` points to an object that doesn't exist anymore. Also, the first call to `Free (I)` will reclaim the storage that was allocated for the object that `I` originally referred to. The call to `Free (I_2)` will then try to reclaim the previously-reclaimed object, but it'll fail in an undefined manner.

Because of these potential errors, you should be very careful when using unchecked deallocation: it is the programmer's responsibility to avoid creating dangling references!

For the example we've just seen, we could avoid creating a dangling reference by explicitly assigning `null` to `I_2` to indicate that it doesn't point to any specific object:

Listing 1382: show_unchecked_deallocation.adb

```

1 with Integer_Types; use Integer_Types;
2
3 procedure Show_Unchecked_Deallocation is
4
5     I, I_2 : Integer_Access;
6
7 begin
8     I := new Integer;
9
10    I_2 := I;
11
12    -- NOTE: I_2 points to the same
13    --       object as I.
14
15    --
16    -- Use I and I_2...
17    --
18    -- ... then deallocate memory...
19    --
20
21    I_2 := null;
22
23    -- NOTE: now, I_2 doesn't point to
24    --       any object, so calling
25    --       Free (I_2) is OK.
26
27    Free (I);
28    Free (I_2);
29 end Show_Unchecked_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Unchecked_Deallocation
 MD5: 3381ba594cbbc0f1547e3f819bae0f97

Now, calling `Free (I_2)` doesn't cause any issues because it doesn't point to any object.

Note, however, that this code example is just meant to illustrate the issues of dangling pointers and how we could circumvent them. We're not suggesting to use this approach when designing an implementation. In fact, it's not practical for the programmer to make every possible dangling reference become null if the calls to `Free` are strewn throughout the code.

The suggested design is to not use `Free` in the client code, but instead hide its use within bigger abstractions. In that way, all the occurrences of the calls to `Free` are in one package, and the programmer of that package can then prevent dangling references. We'll discuss these *design strategies* (page 956) later on.

Dereferencing dangling references

Of course, you shouldn't try to dereference a dangling reference because your program becomes erroneous, as we discuss in this section. Let's see an example:

Listing 1383: show_unchecked_deallocation.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Integer_Types; use Integer_Types;
3

```

(continues on next page)

(continued from previous page)

```

4  procedure Show_Unchecked_Deallocation is
5
6      I_1, I_2 : Integer_Access;
7
8  begin
9      I_1 := new Integer'(42);
10     I_2 := I_1;
11
12     Put_Line ("I_1.all = "
13              & Integer'Image (I_1.all));
14     Put_Line ("I_2.all = "
15              & Integer'Image (I_2.all));
16
17     Put_Line ("Freeing I_1");
18     Free (I_1);
19
20     if I_1 /= null then
21         Put_Line ("I_1.all = "
22                  & Integer'Image (I_1.all));
23     end if;
24
25     if I_2 /= null then
26         Put_Line ("I_2.all = "
27                  & Integer'Image (I_2.all));
28     end if;
29 end Show_Unchecked_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_Deallocation.Unchecked_Deallocation
MD5: 8536190aa5bbafa715ad8153aaeb4889

Runtime output

```

I_1.all = 42
I_2.all = 42
Freeing I_1
I_2.all = -918074818

```

In this example, we allocate an object for `I_1` and make `I_2` point to the same object. Then, we call `Free (I)`, which has the following consequences:

- The call to `Free (I_1)` will try to reclaim the storage for the original object (`I_1.all`), so it may be reused for other allocations.
- `I_1 = null` after the call to `Free (I_1)`.
- `I_2` becomes a dangling reference by the call to `Free (I_1)`.
 - In other words, `I_2` is still non-null, and what it points to is now undefined.

In principle, we could check for `null` before trying to dereference the access value. (Remember that when deallocating an object via a call to `Free`, the corresponding access value is set to `null`.) In fact, this strategy works fine for `I_1`, but it doesn't work for `I_2` because the access value is not `null`. As a consequence, the application tries to dereference `I_2`.

Dereferencing a dangling reference is erroneous: the behavior is undefined in this case. For the example we've just seen,

- `I_2.all` might make the application crash;
- `I_2.all` might give us a different value than before;

- `I_2.all` might even give us the same value as before (42) if the original object is still available.

Because the effect is unpredictable, it might be really difficult to debug the application and identify the cause.

Having dangling pointers in an application should be avoided at all costs! Again, it is the programmer's responsibility to be very careful when using unchecked deallocation: avoid creating dangling references!

i In the Ada Reference Manual

- 13.9.1 Data Validity²⁶⁷
- 13.11.2 Unchecked Storage Deallocation²⁶⁸

Restrictions for `Ada.Unchecked_Deallocation`

There are two unsurprising restrictions for `Ada.Unchecked_Deallocation`:

1. It cannot be instantiated for access-to-constant types; and
2. It cannot be used when the `Storage_Size` aspect of a type is zero (i.e. when its storage pool is empty).

(Note that this last restriction also applies to the allocation via `new`.)

Let's see an example of these restrictions:

Listing 1384: `show_unchecked_deallocation_errors.adb`

```

1  with Ada.Unchecked_Deallocation;
2
3  procedure Show_Unchecked_Deallocation_Errors is
4
5      type Integer_Access_Zero is access Integer
6          with Storage_Size => 0;
7
8      procedure Free is
9          new Ada.Unchecked_Deallocation
10             (Object => Integer,
11              Name   => Integer_Access_Zero);
12
13      type Constant_Integer_Access is
14          access constant Integer;
15
16      -- ERROR: Cannot use access-to-constant type
17      --          for Name
18      procedure Free is
19          new Ada.Unchecked_Deallocation
20             (Object => Integer,
21              Name   => Constant_Integer_Access);
22
23      I : Integer_Access_Zero;
24
25  begin
26      -- ERROR: Cannot allocate objects from
27      --          empty storage pool
28      I := new Integer;
29

```

(continues on next page)

²⁶⁷ <http://www.ada-auth.org/standards/22rm/html/RM-13-9-1.html>

²⁶⁸ <http://www.ada-auth.org/standards/22rm/html/RM-13-11-2.html>

(continued from previous page)

```

30  -- ERROR: Cannot deallocate objects from
31  --      empty storage pool
32  Free (I);
33  end Show_Unchecked_Deallocation_Errors;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Unchecked_
 ↳Deallocation.Unchecked_Deallocation_Error
 MD5: 5032d13b2eb6b7ca1979282ddd6df98a

Build output

```

show_unchecked_deallocation_errors.adb:21:19: error: actual type must be access-to-
↳variable type
show_unchecked_deallocation_errors.adb:21:19: error: instantiation abandoned
show_unchecked_deallocation_errors.adb:28:09: error: allocation from empty storage_
↳pool
show_unchecked_deallocation_errors.adb:32:04: error: deallocation from empty_
↳storage pool
gprbuild: *** compilation phase failed

```

Here, we see that trying to instantiate `Ada.Unchecked_Deallocation` for the `Constant_Integer_Access` type is rejected by the compiler. Similarly, we cannot allocate or deallocate an object for the `Integer_Access_Zero` type because its storage pool is empty.

28.1.13 Null & Not Null Access

Note

This section was originally written by Robert A. Duff and published as [Gem #23: Null Considered Harmful](#)²⁶⁹ and [Gem #24](#)²⁷⁰.

Ada, like many languages, defines a special **null** value for access types. All values of an access type designate some object of the designated type, except for **null**, which does not designate any object. The null value can be used as a special flag. For example, a singly-linked list can be null-terminated. A Lookup function can return **null** to mean "not found", presuming the result is of an access type:

Listing 1385: `show_null_return.ads`

```

1  package Show_Null_Return is
2
3      type Ref_Element is access all Element;
4
5      Not_Found : constant Ref_Element := null;
6
7      function Lookup (T : Table) return Ref_Element;
8      -- Returns Not_Found if not found.
9  end Show_Null_Return;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
 ↳Access.Null_Return
 MD5: 6c4eed750d42685198ec9495805e3e23

²⁶⁹ <https://www.adacore.com/gems/ada-gem-23>

²⁷⁰ <https://www.adacore.com/gems/ada-gem-24>

An alternative design for Lookup would be to raise an exception:

Listing 1386: show_not_found_exception.ads

```

1 package Show_Not_Found_Exception is
2   Not_Found : exception;
3
4   function Lookup (T : Table) return Ref_Element;
5   -- Raises Not_Found if not found.
6   -- Never returns null.
7 end Show_Not_Found_Exception;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↳ Access.Not_Found_Exception
MD5: 6ef47b32d4923838ffc28f43e5db323c

Neither design is better in all situations; it depends in part on whether we consider the "not found" situation to be exceptional.

Clearly, the client calling Lookup needs to know whether it can return **null**, and if so, what that means. In general, it's a good idea to document whether things can be null or not, especially for formal parameters and function results. Prior to Ada 2005, we would do that with comments. Since Ada 2005, we can use the **not null** syntax:

Listing 1387: show_not_null_return.ads

```

1 package Show_Not_Null_Return is
2   type Ref_Element is access all Element;
3
4   Not_Found : constant Ref_Element := null;
5
6   function Lookup (T : Table)
7     return not null Ref_Element;
8   -- Possible since Ada 2005.
9 end Show_Not_Null_Return;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
↳ Access.Not_Null_Return
MD5: 4c0bb95da3b5a7c555a763c4951f7e21

This is a complete package for the code snippets above:

Listing 1388: example.ads

```

1 package Example is
2
3   type Element is limited private;
4   type Ref_Element is access all Element;
5
6   type Table is limited private;
7
8   Not_Found : constant Ref_Element := null;
9   function Lookup (T : Table)
10     return Ref_Element;
11   -- Returns Not_Found if not found.
12
13   Not_Found_2 : exception;
14   function Lookup_2 (T : Table)
```

(continues on next page)

(continued from previous page)

```

15         return not null Ref_Element;
16     -- Raises Not_Found_2 if not found.
17
18     procedure P (X : not null Ref_Element);
19
20     procedure Q (X : not null Ref_Element);
21
22 private
23     type Element is limited
24         record
25             Component : Integer;
26         end record;
27     type Table is limited null record;
28 end Example;

```

Listing 1389: example.adb

```

1 package body Example is
2
3     An_Element : aliased Element;
4
5     function Lookup (T : Table)
6         return Ref_Element is
7         pragma Unreferenced (T);
8     begin
9         -- ...
10        return Not_Found;
11    end Lookup;
12
13    function Lookup_2 (T : Table)
14        return not null Ref_Element
15    is
16    begin
17        -- ...
18        raise Not_Found_2;
19
20        return An_Element'Access;
21        -- suppress error: 'missing "return"
22        -- statement in function body'
23    end Lookup_2;
24
25    procedure P (X : not null Ref_Element) is
26    begin
27        X.all.Component := X.all.Component + 1;
28    end P;
29
30    procedure Q (X : not null Ref_Element) is
31    begin
32        for I in 1 .. 1000 loop
33            P (X);
34        end loop;
35    end Q;
36
37    procedure R is
38    begin
39        Q (An_Element'Access);
40    end R;
41
42    pragma Unreferenced (R);
43
44 end Example;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_↪Access.Complete_Null_Return
MD5: 01895c7d5f843fd215dcc21d807d4187

In general, it's better to use the language proper for documentation, when possible, rather than comments, because compile-time and/or run-time checks can help ensure that the "documentation" is actually true. With comments, there's a greater danger that the comment will become false during maintenance, and false documentation is obviously a menace.

In many, perhaps most cases, **null** is just a tripping hazard. It's a good idea to put in **not null** when possible. In fact, a good argument can be made that **not null** should be the default, with extra syntax required when **null** is wanted. This is the way *Standard ML*²⁷¹ works, for example — you don't get any special null-like value unless you ask for it. Of course, because Ada 2005 needs to be compatible with previous versions of the language, **not null** cannot be the default for Ada.

One word of caution: access objects are default-initialized to **null**, so if you have a **not null** object (or component) you had better initialize it explicitly, or you will get *Constraint_Error*. **not null** is more often useful on parameters and function results, for this reason.

Another advantage of **not null** over comments is for efficiency. Consider procedures P and Q in this example:

Listing 1390: example-processing.ads

```
1 package Example.Processing is
2
3     procedure P (X : not null Ref_Element);
4
5     procedure Q (X : not null Ref_Element);
6
7 end Example.Processing;
```

Listing 1391: example-processing.adb

```
1 package body Example.Processing is
2
3     procedure P (X : not null Ref_Element) is
4     begin
5         X.all.Component := X.all.Component + 1;
6     end P;
7
8     procedure Q (X : not null Ref_Element) is
9     begin
10         for I in 1 .. 1000 loop
11             P (X);
12         end loop;
13     end Q;
14
15 end Example.Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_↪Access.Complete_Null_Return
MD5: dc34b1a27737d57c041be6260dd577fd

²⁷¹ https://en.wikipedia.org/wiki/Standard_ML

Without **not null**, the generated code for P will do a check that $X \neq \text{null}$, which may be costly on some systems. P is called in a loop, so this check will likely occur many times. With **not null**, the check is pushed to the call site. Pushing checks to the call site is usually beneficial because

1. the check might be hoisted out of a loop by the optimizer, or
2. the check might be eliminated altogether, as in the example above, where the compiler knows that `An_Element'Access` cannot be **null**.

This is analogous to the situation with other run-time checks, such as array bounds checks:

Listing 1392: show_process_array.ads

```
1 package Show_Process_Array is
2
3     type My_Index is range 1 .. 10;
4     type My_Array is array (My_Index) of Integer;
5
6     procedure Process_Array
7         (X      : in out My_Array;
8          Index  :      My_Index);
9
10 end Show_Process_Array;
```

Listing 1393: show_process_array.adb

```
1 package body Show_Process_Array is
2
3     procedure Process_Array
4         (X      : in out My_Array;
5          Index  :      My_Index) is
6     begin
7         X (Index) := X (Index) + 1;
8     end Process_Array;
9
10 end Show_Process_Array;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Null_And_Not_Null_
Access.Process_Array
MD5: 32424432f5b2e3013292680f92a04320

If $X \text{ (Index)}$ occurs inside `Process_Array`, there is no need to check that `Index` is in range, because the check is pushed to the caller.

28.1.14 Design strategies for access types

Previously, we learned about *dangling references* (page 939) and discussed the effects of *dereferencing them* (page 949). Also, we've seen the relationship between *unchecked deallocation and dangling references* (page 947). Ensuring that all calls to `Free` for a specific access type will never cause dangling references can become an arduous task — if not impossible — if those calls are located in different parts of the source code.

Although we used access types directly in the main application in many of the previous code examples from this chapter, this approach was in fact selected just for illustration purposes — i.e. to make the code look simpler. In general, however, we should avoid this approach. Instead, our recommendation is to encapsulate the access types in some form of abstraction. In this section, we discuss design strategies for access types that take this recommendation into account.

Abstract data type for access types

The simplest form of abstraction is of course an abstract data type. For example, we could declare a limited private type, which allows us to hide the access type and to avoid copies of references that could potentially become dangling references. (We discuss limited private types later *in another chapter* (page 1071).)

Let's see an example:

Listing 1394: access_type_abstraction.ads

```

1 package Access_Type_Abstraction is
2
3     type Info is limited private;
4
5     function To_Info (S : String) return Info;
6
7     function To_String (Obj : Info)
8         return String;
9
10    function Copy (Obj : Info) return Info;
11
12    procedure Copy (To   : in out Info;
13                  From :      Info);
14
15    procedure Append (Obj : in out Info;
16                    S   : String);
17
18    procedure Reset (Obj : in out Info);
19
20    procedure Destroy (Obj : in out Info);
21
22 private
23
24     type Info is access String;
25
26 end Access_Type_Abstraction;
```

Listing 1395: access_type_abstraction.adb

```

1 with Ada.Unchecked_Deallocation;
2
3 package body Access_Type_Abstraction is
4
5     function To_Info (S : String) return Info is
6         (new String'(S));
7
8     function To_String (Obj : Info)
9         return String is
10         (if Obj /= null then Obj.all else "");
11
12     function Copy (Obj : Info) return Info is
13         (To_Info (To_String (Obj)));
14
15     procedure Copy (To   : in out Info;
16                   From :      Info) is
17     begin
18         Destroy (To);
19         To := Copy (From);
20     end Copy;
21
22     procedure Append (Obj : in out Info;
```

(continues on next page)

(continued from previous page)

```

23         S : String) is
24     New_Info : constant Info :=
25         To_Info (To_String (Obj) & S);
26     begin
27         Destroy (Obj);
28         Obj := New_Info;
29     end Append;
30
31     procedure Reset (Obj : in out Info) is
32     begin
33         Destroy (Obj);
34     end Reset;
35
36     procedure Destroy (Obj : in out Info) is
37     procedure Free is
38         new Ada.Unchecked_Deallocation
39             (Object => String,
40              Name   => Info);
41     begin
42         Free (Obj);
43     end Destroy;
44
45 end Access_Type_Abstraction;

```

Listing 1396: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_Type_Abstraction;
4  use Access_Type_Abstraction;
5
6  procedure Main is
7      Obj_1 : Info := To_Info ("hello");
8      Obj_2 : Info := Copy (Obj_1);
9  begin
10     Put_Line ("TO_INFO / COPY");
11     Put_Line ("Obj_1 : "
12             & To_String (Obj_1));
13     Put_Line ("Obj_2 : "
14             & To_String (Obj_2));
15     Put_Line ("-----");
16
17     Reset (Obj_1);
18     Append (Obj_2, " world");
19
20     Put_Line ("RESET / APPEND");
21     Put_Line ("Obj_1 : "
22             & To_String (Obj_1));
23     Put_Line ("Obj_2 : "
24             & To_String (Obj_2));
25     Put_Line ("-----");
26
27     Copy (From => Obj_2,
28          To   => Obj_1);
29
30     Put_Line ("COPY");
31     Put_Line ("Obj_1 : "
32             & To_String (Obj_1));
33     Put_Line ("Obj_2 : "
34             & To_String (Obj_2));
35     Put_Line ("-----");

```

(continues on next page)

(continued from previous page)

```

36
37   Destroy (Obj_1);
38   Destroy (Obj_2);
39
40   Put_Line ("DESTROY");
41   Put_Line ("Obj_1 : "
42           & To_String (Obj_1));
43   Put_Line ("Obj_2 : "
44           & To_String (Obj_2));
45   Put_Line ("-----");
46
47   Append (Obj_1, "hey");
48
49   Put_Line ("APPEND");
50   Put_Line ("Obj_1 : "
51           & To_String (Obj_1));
52   Put_Line ("-----");
53
54   Put_Line ("APPEND");
55   Append (Obj_1, " there");
56   Put_Line ("Obj_1 : "
57           & To_String (Obj_1));
58
59   Destroy (Obj_1);
60   Destroy (Obj_2);
61 end Main;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Design_Strategies.
↳ Access_Type_Abstraction
MD5: a335caeba4f1fb952a2e0d8d6bc52f75

Runtime output

```

TO_INFO / COPY
Obj_1 : hello
Obj_2 : hello
-----
RESET / APPEND
Obj_1 :
Obj_2 : hello world
-----
COPY
Obj_1 : hello world
Obj_2 : hello world
-----
DESTROY
Obj_1 :
Obj_2 :
-----
APPEND
Obj_1 : hey
-----
APPEND
Obj_1 : hey there
```

In this example, we hide an access type in the Info type — a limited private type. We allocate an object of this type in the To_Info function and deallocate it in the Destroy procedure. Also, we make sure that the reference isn't copied in the Copy function — we only copy the designated value in this function. This strategy eliminates the possibility of

dangling references, as each reference is encapsulated in an object of Info type.

Controlled type for access types

In the previous code example, the Destroy procedure had to be called to deallocate the hidden access object. We could make sure that this deallocation happens automatically by using a controlled (or limited controlled) type. (We discuss *controlled types* (page 1118) in another chapter.)

Let's adapt the previous example and declare Info as a limited controlled type:

Listing 1397: access_type_abstraction.ads

```
1 with Ada.Finalization;
2
3 package Access_Type_Abstraction is
4
5     type Info is limited private;
6
7     function To_Info (S : String) return Info;
8
9     function To_String (Obj : Info)
10         return String;
11
12     function Copy (Obj : Info) return Info;
13
14     procedure Copy (To : in out Info;
15                   From : Info);
16
17     procedure Append (Obj : in out Info;
18                    S : String);
19
20     procedure Reset (Obj : in out Info);
21
22 private
23
24     type String_Access is access String;
25
26     type Info is new
27         Ada.Finalization.Limited_Controlled with
28         record
29             Str_A : String_Access;
30         end record;
31
32     procedure Initialize (Obj : in out Info);
33     procedure Finalize (Obj : in out Info);
34
35 end Access_Type_Abstraction;
```

Listing 1398: access_type_abstraction.adb

```
1 with Ada.Unchecked_Deallocation;
2
3 package body Access_Type_Abstraction is
4
5     --
6     --  STRING_ACCESS SUBPROGRAMS
7     --
8
9     function To_String_Access (S : String)
10         return String_Access
11 is
```

(continues on next page)

(continued from previous page)

```

12     (new String'(S));
13
14     function To_String (S : String_Access)
15         return String is
16     (if S /= null then S.all else "");
17
18     procedure Free is
19     new Ada.Unchecked_Deallocation
20     (Object => String,
21      Name  => String_Access);
22
23     --
24     --  PRIVATE SUBPROGRAMS
25     --
26
27     procedure Initialize (Obj : in out Info) is
28     begin
29         -- Put_Line ("Initializing Info");
30         Obj.Str_A := null;
31         ~~~~~
32         -- NOTE: This line has just been added to
33         --       illustrate the "automatic" call to
34         --       Initialize. Actually, this
35         --       assignment isn't needed, as
36         --       the Str_A component is
37         --       automatically initialized to null
38         --       upon object construction.
39     end Initialize;
40
41     procedure Finalize (Obj : in out Info) is
42     begin
43         -- Put_Line ("Finalizing Info");
44         Free (Obj.Str_A);
45     end Finalize;
46
47     --
48     --  PUBLIC SUBPROGRAMS
49     --
50
51     function To_Info (S : String) return Info is
52     (Ada.Finalization.Limited_Controlled
53      with Str_A => To_String_Access (S));
54
55     function To_String (Obj : Info)
56         return String is
57     (To_String (Obj.Str_A));
58
59     function Copy (Obj : Info) return Info is
60     (To_Info (To_String (Obj.Str_A)));
61
62     procedure Copy (To   : in out Info;
63                   From :      Info) is
64     begin
65         Free (To.Str_A);
66         To.Str_A := To_String_Access
67             (To_String (From.Str_A));
68     end Copy;
69
70     procedure Append (Obj : in out Info;
71                     S   :      String) is
72         New_Str_A : constant String_Access :=

```

(continues on next page)

(continued from previous page)

```

73         To_String_Access
74         (To_String (Obj.Str_A) & S);
75     begin
76         Free (Obj.Str_A);
77         Obj.Str_A := New_Str_A;
78     end Append;
79
80     procedure Reset (Obj : in out Info) is
81     begin
82         Free (Obj.Str_A);
83     end Reset;
84
85 end Access_Type_Abstraction;

```

Listing 1399: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_Type_Abstraction;
4  use Access_Type_Abstraction;
5
6  procedure Main is
7      Obj_1 : Info := To_Info ("hello");
8      Obj_2 : Info := Copy (Obj_1);
9  begin
10     --
11     --  TO_INFO / COPY
12     --
13     Put_Line ("TO_INFO / COPY");
14
15     Put_Line ("Obj_1 : "
16             & To_String (Obj_1));
17     Put_Line ("Obj_2 : "
18             & To_String (Obj_2));
19     Put_Line ("-----");
20
21     --
22     --  RESET: Obj_1
23     --  APPEND: Obj_2
24     --
25     Put_Line ("RESET / APPEND");
26
27     Reset (Obj_1);
28     Append (Obj_2, " world");
29
30     Put_Line ("Obj_1 : "
31             & To_String (Obj_1));
32     Put_Line ("Obj_2 : "
33             & To_String (Obj_2));
34     Put_Line ("-----");
35
36     --
37     --  COPY: Obj_2 => Obj_1
38     --
39     Put_Line ("COPY");
40
41     Copy (From => Obj_2,
42          To   => Obj_1);
43
44     Put_Line ("Obj_1 : "
45             & To_String (Obj_1));

```

(continues on next page)

(continued from previous page)

```

46  Put_Line ("Obj_2 : "
47          & To_String (Obj_2));
48  Put_Line ("-----");
49
50  --
51  --  RESET: Obj_1, Obj_2
52  --
53  Put_Line ("RESET");
54
55  Reset (Obj_1);
56  Reset (Obj_2);
57
58  Put_Line ("Obj_1 : "
59          & To_String (Obj_1));
60  Put_Line ("Obj_2 : "
61          & To_String (Obj_2));
62  Put_Line ("-----");
63
64  --
65  --  COPY: Obj_2 => Obj_1
66  --
67  Put_Line ("COPY");
68
69  Copy (From => Obj_2,
70       To   => Obj_1);
71
72  Put_Line ("Obj_1 : "
73          & To_String (Obj_1));
74  Put_Line ("Obj_2 : "
75          & To_String (Obj_2));
76  Put_Line ("-----");
77
78  --
79  --  APPEND: Obj_1 with "hey"
80  --
81  Put_Line ("APPEND");
82
83  Append (Obj_1, "hey");
84
85  Put_Line ("Obj_1 : "
86          & To_String (Obj_1));
87  Put_Line ("-----");
88
89  --
90  --  APPEND: Obj_1 with "there"
91  --
92  Put_Line ("APPEND");
93
94  Append (Obj_1, " there");
95
96  Put_Line ("Obj_1 : "
97          & To_String (Obj_1));
98  end Main;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Design_Strategies.
 ↪ Access_Type_Limited_Controlled_Abstraction
 MD5: e98659ad1b87be56fb173fa407ab7e82

Runtime output

```
TO_INFO / COPY
Obj_1 : hello
Obj_2 : hello
-----
RESET / APPEND
Obj_1 :
Obj_2 : hello world
-----
COPY
Obj_1 : hello world
Obj_2 : hello world
-----
RESET
Obj_1 :
Obj_2 :
-----
COPY
Obj_1 :
Obj_2 :
-----
APPEND
Obj_1 : hey
-----
APPEND
Obj_1 : hey there
```

Of course, because we're using the `Limited_Controlled` type from the `Ada.Finalization` package, we had to adapt the prototype of the subprograms from the `Access_Type_Abstraction`. In this version of the code, we only have the allocation taking place in the `To_Info` procedure, but we don't have a `Destroy` procedure for deallocation: this call was moved to the `Finalize` procedure.

Since objects of the `Info` type — such as `Obj_1` in the `Show_Access_Type_Abstraction` procedure — are now controlled, the `Finalize` procedure is automatically called when they go out of scope. In this procedure, which we override for the `Info` type, we perform the deallocation of the internal access object `Str_A`. (You may uncomment the calls to `Put_Line` in the body of the `Initialize` and `Finalize` subprograms to confirm that these subprograms are called in the background.)

28.1.15 Access to subprograms

So far in this chapter, we focused mainly on access-to-objects. However, we can use access types to subprograms. This is the topic of this section.

Static vs. dynamic calls

In a typical subprogram call, we indicate the subprogram we want to call statically. For example, let's say we've implemented a procedure `Proc` that calls a procedure `P`:

Listing 1400: `p.ads`

```
1 procedure P (I : in out Integer);
```

Listing 1401: `p.adb`

```
1 procedure P (I : in out Integer) is
2 begin
3   null;
4 end P;
```

Listing 1402: proc.adb

```

1 with P;
2
3 procedure Proc is
4     I : Integer := 0;
5 begin
6     P (I);
7 end Proc;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳Subprograms.Subprogram_Call
 MD5: 0e9547e53d0d02d39920f4d1d6787af6

The call to P is statically dispatched: every time Proc runs and calls P, that call is always to the same procedure. In other words, we can determine at compilation time which procedure is called.

In contrast, an access to a subprogram allows us to dynamically indicate which subprogram we want to call. For example, if we change Proc in the code above to receive the access to a subprogram P as a parameter, the actual procedure that would be called when running Proc would be determined at run time, and it might be different for every call to Proc. In this case, we wouldn't be able to determine at compilation time which procedure would be called in every case. (In some cases, however, it could still be possible to determine which procedure is called by analyzing the argument that is passed to Proc.)

Access to subprogram declaration

We declare an access to a subprogram as a type by writing **access procedure** or **access function** and the corresponding prototype:

Listing 1403: access_to_subprogram_types.ads

```

1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
5
6     type Access_To_Function is
7         access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳Subprograms.Access_To_Subprogram_Types
 MD5: 5f834c1b2044ba5ea7d4835c3ebdedb1

In the designated profile of the access type declarations, we list all the parameters that we expect in the subprogram.

We can use those types to declare access to subprograms — as subprogram parameters, for example:

Listing 1404: access_to_subprogram_params.ads

```

1 with Access_To_Subprogram_Types;
2 use Access_To_Subprogram_Types;

```

(continues on next page)

(continued from previous page)

```

3
4 package Access_To_Subprogram_Params is
5
6     procedure Proc (P : Access_To_Procedure);
7
8 end Access_To_Subprogram_Params;
```

Listing 1405: access_to_subprogram_params.adb

```

1 package body Access_To_Subprogram_Params is
2
3     procedure Proc (P : Access_To_Procedure) is
4         I : Integer := 0;
5     begin
6         P (I);
7         -- P.all (I);
8     end Proc;
9
10 end Access_To_Subprogram_Params;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳ Subprograms.Access_To_Subprogram_Types
MD5: 17c1a07f48d9fb0efef37aa4c5ec8a51

In the implementation of the Proc procedure of the code example, we call the P procedure by simply passing I as a parameter. In this case, P is automatically dereferenced. We may, however, explicitly dereference P by writing P.all (I).

Before we use this package, let's implement a simple procedure that we'll use later on:

Listing 1406: add_ten.ads

```

1 procedure Add_Ten (I : in out Integer);
```

Listing 1407: add_ten.adb

```

1 procedure Add_Ten (I : in out Integer) is
2 begin
3     I := I + 10;
4 end Add_Ten;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳ Subprograms.Access_To_Subprogram_Types
MD5: 8553ad7329bf1ed727147b47b7355a70

Now, we can get access to a subprogram by using the **Access** attribute and pass it as an actual parameter:

Listing 1408: show_access_to_subprograms.adb

```

1 with Access_To_Subprogram_Params;
2 use Access_To_Subprogram_Params;
3
4 with Add_Ten;
5
6 procedure Show_Access_To_Subprograms is
```

(continues on next page)

(continued from previous page)

```

7 begin
8   Proc (Add_Ten'Access);
9   --      ^ Getting access to Add_Ten
10  --      procedure and passing it
11  --      to Proc
12 end Show_Access_To_Subprograms;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
 MD5: 599e9d1306da48e3c532692b34c02a1d

Here, we get access to the Add_Ten procedure and pass it to the Proc procedure.

i In the Ada Reference Manual

- 3.10 Access Types²⁷²

Objects of access-to-subprogram type

In the previous example, the Proc procedure had a parameter of access-to-subprogram type. In addition to parameters, we can of course declare *objects* of access-to-subprogram types as well. For example, we can extend our previous test application and declare an object P of access-to-subprogram type. Before we do so, however, let's implement another small procedure that we'll use later on:

Listing 1409: add_twenty.ads

```
1 procedure Add_Twenty (I : in out Integer);
```

Listing 1410: add_twenty.adb

```

1 procedure Add_Twenty (I : in out Integer) is
2 begin
3   I := I + 20;
4 end Add_Twenty;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
 MD5: 697959b806f6f2bfba248ec15c47883b

In addition to Add_Ten, we've implemented the Add_Twenty procedure, which we use in our extended test application:

Listing 1411: show_access_to_subprograms.adb

```

1 with Access_To_Subprogram_Types;
2 use Access_To_Subprogram_Types;
3
4 with Access_To_Subprogram_Params;
5 use Access_To_Subprogram_Params;
6
7 with Add_Ten;
```

(continues on next page)

²⁷² <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

8  with Add_Twenty;
9
10 procedure Show_Access_To_Subprograms is
11     P      : Access_To_Procedure;
12     Some_Int : Integer := 0;
13 begin
14     P := Add_Ten'Access;
15     --      ^ Getting access to Add_Ten
16     --      procedure and assigning it
17     --      to P
18
19     Proc (P);
20     --      ^ Passing access-to-subprogram as an
21     --      actual parameter
22
23     P (Some_Int);
24     --      ^ Using access-to-subprogram object in a
25     --      subprogram call
26
27     P := Add_Twenty'Access;
28     --      ^ Getting access to Add_Twenty
29     --      procedure and assigning it
30     --      to P
31
32     Proc (P);
33     P (Some_Int);
34 end Show_Access_To_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
MD5: 7b4ea19187806e88ba65847876cafb4f

In the `Show_Access_To_Subprograms` procedure, we see the declaration of our access-to-subprogram object `P` (of `Access_To_Procedure` type). We get access to the `Add_Ten` procedure and assign it to `P`, and we then do the same for the `Add_Twenty` procedure.

We can use an access-to-subprogram object either as the actual parameter of a subprogram call, or in a subprogram call. In the code example, we're passing `P` as the actual parameter of the `Proc` procedure in the `Proc (P)` calls. Also, we're calling the subprogram assigned to (designated by the current value of) `P` in the `P (Some_Int)` calls.

Components of access-to-subprogram type

In addition to declaring subprogram parameters and objects of access-to-subprogram types, we can declare components of these types. For example:

Listing 1412: `access_to_subprogram_types.ads`

```

1  package Access_To_Subprogram_Types is
2
3      type Access_To_Procedure is
4          access procedure (I : in out Integer);
5
6      type Access_To_Function is
7          access function (I : Integer) return Integer;
8
9      type Access_To_Procedure_Array is
10         array (Positive range <>) of

```

(continues on next page)

(continued from previous page)

```

11     Access_To_Procedure;
12
13     type Access_To_Function_Array is
14         array (Positive range <>) of
15             Access_To_Function;
16
17     type Rec_Access_To_Procedure is record
18         AP : Access_To_Procedure;
19     end record;
20
21     type Rec_Access_To_Function is record
22         AF : Access_To_Function;
23     end record;
24
25 end Access_To_Subprogram_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
MD5: 32203838b97af66ef6ca3f6b1ce646a5

Here, the access-to-procedure type `Access_To_Procedure` is used as a component of the array type `Access_To_Procedure_Array` and the record type `Rec_Access_To_Procedure`. Similarly, the access-to-function type `Access_To_Function` type is used as a component of the array type `Access_To_Function_Array` and the record type `Rec_Access_To_Function`.

Let's see two test applications using these types. First, let's use the `Access_To_Procedure_Array` array type in a test application:

Listing 1413: show_access_to_subprograms.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_To_Subprogram_Types;
4  use Access_To_Subprogram_Types;
5
6  with Add_Ten;
7  with Add_Twenty;
8
9  procedure Show_Access_To_Subprograms is
10     PA : constant
11         Access_To_Procedure_Array (1 .. 2) :=
12         (Add_Ten'Access,
13          Add_Twenty'Access);
14
15     Some_Int : Integer := 0;
16 begin
17     Put_Line ("Some_Int: " & Some_Int'Image);
18
19     for I in PA'Range loop
20         PA (I) (Some_Int);
21         Put_Line ("Some_Int: " & Some_Int'Image);
22     end loop;
23 end Show_Access_To_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
MD5: f1d10056b4b3424bd30d954f34caa255

Runtime output

```
Some_Int:  0
Some_Int: 10
Some_Int: 30
```

Here, we declare the PA array and use the access to the Add_Ten and Add_Twenty procedures as its components. We can call any of these procedures by simply specifying the index of the component, e.g. PA (2). Once we specify the procedure we want to use, we simply pass the parameters, e.g.: PA (2) (Some_Int).

Now, let's use the Rec_Access_To_Procedure record type in a test application:

Listing 1414: show_access_to_subprograms.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 with Add_Ten;
7 with Add_Twenty;
8
9 procedure Show_Access_To_Subprograms is
10     RA      : Rec_Access_To_Procedure;
11     Some_Int : Integer := 0;
12 begin
13     Put_Line ("Some_Int: " & Some_Int'Image);
14
15     RA := (AP => Add_Ten'Access);
16     RA.AP (Some_Int);
17     Put_Line ("Some_Int: " & Some_Int'Image);
18
19     RA := (AP => Add_Twenty'Access);
20     RA.AP (Some_Int);
21     Put_Line ("Some_Int: " & Some_Int'Image);
22 end Show_Access_To_Subprograms;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳Subprograms.Access_To_Subprogram_Types
MD5: 4b23b5f6a8c252a1a014a2b54fa32c1a
```

Runtime output

```
Some_Int:  0
Some_Int: 10
Some_Int: 30
```

Here, we declare two record aggregates where we specify the AP component, e.g.: (AP => Add_Ten'Access), which indicates the access-to-subprogram we want to use. We can call the subprogram by simply accessing the AP component, i.e.: RA.AP.

Access-to-subprogram as discriminant types

As you might expect, we can use access-to-subprogram types when declaring discriminants. In fact, when we were talking about *discriminants as access values* (page 891) earlier on, we used access-to-object types in our code examples, but we could have used access-to-subprogram types as well. For example:

Listing 1415: custom_processing.ads

```

1 package Custom_Processing is
2
3   -- Declaring an access type:
4   type Integer_Processing is
5     access procedure (I : in out Integer);
6
7   -- Declaring a discriminant with this
8   -- access type:
9   type Rec (IP : Integer_Processing) is
10     private;
11
12   procedure Init (R      : in out Rec;
13                  Value :      Integer);
14
15   procedure Process (R : in out Rec);
16
17   procedure Show (R : Rec);
18
19 private
20
21   type Rec (IP : Integer_Processing) is
22     record
23       I : Integer := 0;
24     end record;
25
26 end Custom_Processing;
```

Listing 1416: custom_processing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Processing is
4
5   procedure Init (R      : in out Rec;
6                  Value :      Integer) is
7   begin
8     R.I := Value;
9   end Init;
10
11   procedure Process (R : in out Rec) is
12   begin
13     R.IP (R.I);
14     -- ^^^^^
15     -- Calling procedure that we specified as
16     -- the record's discriminant
17   end Process;
18
19   procedure Show (R : Rec) is
20   begin
21     Put_Line ("R.I = "
22              & Integer'Image (R.I));
23   end Show;
24
25 end Custom_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳ Subprograms.Access_To_Subprogram_Types
 MD5: 02fc0c51722c321c4ec6115de68d1c06

In this example, we declare the access-to-subprogram type `Integer_Processing`, which we use as the IP discriminant of the `Rec` type. In the `Process` procedure, we call the IP procedure that we specified as the record's discriminant (`R.IP (R.I)`).

Before we look at a test application for this package, let's implement another small procedure:

Listing 1417: mult_two.ads

```
1 procedure Mult_Two (I : in out Integer);
```

Listing 1418: mult_two.adb

```
1 procedure Mult_Two (I : in out Integer) is
2 begin
3     I := I * 2;
4 end Mult_Two;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳Subprograms.Access_To_Subprogram_Types
MD5: cd43fa39dac9a1c9182f69d32eab1d26
```

Now, let's look at the test application:

Listing 1419: show_access_to_subprogram_discriminants.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2
3  with Custom_Processing; use Custom_Processing;
4
5  with Add_Ten;
6  with Mult_Two;
7
8  procedure Show_Access_To_Subprogram_Discriminants
9  is
10
11     R_Add_Ten  : Rec (IP => Add_Ten'Access);
12     --          ~~~~~
13     --          Using access-to-subprogram as a
14     --          discriminant
15
16     R_Mult_Two : Rec (IP => Mult_Two'Access);
17     --          ~~~~~
18     --          Using access-to-subprogram as a
19     --          discriminant
20
21  begin
22     Init (R_Add_Ten, 1);
23     Init (R_Mult_Two, 2);
24
25     Put_Line ("---- R_Add_Ten ----");
26     Show (R_Add_Ten);
27
28     Put_Line ("Calling Process procedure...");
29     Process (R_Add_Ten);
30     Show (R_Add_Ten);
31
32     Put_Line ("---- R_Mult_Two ----");
33     Show (R_Mult_Two);
34

```

(continues on next page)

(continued from previous page)

```

35   Put_Line ("Calling Process procedure...");
36   Process (R_Mult_Two);
37   Show (R_Mult_Two);
38 end Show_Access_To_Subprogram_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_To_Subprogram_Types
 MD5: 544c224f8bc8e6ba2db4914c2a3dcff4

Runtime output

```

---- R_Add_Ten ----
R.I = 1
Calling Process procedure...
R.I = 11
---- R_Mult_Two ----
R.I = 2
Calling Process procedure...
R.I = 4

```

In this procedure, we declare the `R_Add_Ten` and `R_Mult_Two` of `Rec` type and specify the access to `Add_Ten` and `Mult_Two`, respectively, as the IP discriminant. The procedure we specified here is then called inside a call to the `Process` procedure.

Access-to-subprograms as formal parameters

We can use access-to-subprograms types when declaring formal parameters. For example, let's revisit the `Custom_Processing` package from the previous section and convert it into a generic package.

Listing 1420: `gen_custom_processing.ads`

```

1  generic
2    type T is private;
3
4    --
5    --   Declaring formal access-to-subprogram
6    --   type:
7    --
8    type T_Processing is
9      access procedure (Element : in out T);
10
11   --
12   --   Declaring formal access-to-subprogram
13   --   parameter:
14   --
15   Proc : T_Processing;
16
17   with function Image_T (Element : T)
18     return String;
19 package Gen_Custom_Processing is
20
21   type Rec is private;
22
23   procedure Init (R      : in out Rec;
24                 Value :      T);
25
26   procedure Process (R : in out Rec);

```

(continues on next page)

(continued from previous page)

```

27
28     procedure Show (R : Rec);
29
30 private
31
32     type Rec is record
33         Comp : T;
34     end record;
35
36 end Gen_Custom_Processing;

```

Listing 1421: gen_custom_processing.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_Custom_Processing is
4
5     procedure Init (R      : in out Rec;
6                    Value :      T) is
7     begin
8         R.Comp := Value;
9     end Init;
10
11     procedure Process (R : in out Rec) is
12     begin
13         Proc (R.Comp);
14     end Process;
15
16     procedure Show (R : Rec) is
17     begin
18         Put_Line ("R.Comp = "
19                 & Image_T (R.Comp));
20     end Show;
21
22 end Gen_Custom_Processing;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
↳ Subprograms.Access_To_Subprogram_Types
MD5: 6f06e066bafa5f02abb3ee1b33ea0831

In this version of the procedure, instead of declaring Proc as a discriminant of the Rec record, we're declaring it as a formal parameter of the Gen_Custom_Processing package. Also, we're declaring an access-to-subprogram type (T_Processing) as a formal parameter. (Note that, in contrast to these two parameters that we've just mentioned, Image_T is not a formal access-to-subprogram parameter: it's actually just a formal subprogram.)

We then instantiate the Gen_Custom_Processing package in our test application:

Listing 1422: show_access_to_subprogram_as_formal_parameter.adb

```

1 with Gen_Custom_Processing;
2
3 with Add_Ten;
4
5 with Ada.Text_IO; use Ada.Text_IO;
6
7 procedure
8     Show_Access_To_Subprogram_As_Formal_Parameter
9 is

```

(continues on next page)

(continued from previous page)

```

10  type Integer_Processing is
11      access procedure (I : in out Integer);
12
13  package Custom_Processing is new
14      Gen_Custom_Processing
15      (T      => Integer,
16       T_Processing => Integer_Processing,
17       --
18       --      access-to-subprogram type
19       Proc      => Add_Ten'Access,
20       --
21       --      access-to-subprogram
22       Image_T    => Integer'Image);
23  use Custom_Processing;
24
25  R_Add_Ten : Rec;
26
27  begin
28      Init (R_Add_Ten, 1);
29
30      Put_Line ("---- R_Add_Ten ----");
31      Show (R_Add_Ten);
32
33      Put_Line ("Calling Process procedure...");
34      Process (R_Add_Ten);
35      Show (R_Add_Ten);
36  end Show_Access_To_Subprogram_As_Formal_Parameter;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳ Subprograms.Access_To_Subprogram_Types
 MD5: 6ae27ebd59e5307551e9a38f3b94c70c

Runtime output

```

---- R_Add_Ten ----
R.Comp = 1
Calling Process procedure...
R.Comp = 11

```

Here, we instantiate the `Gen_Custom_Processing` package as `Custom_Processing` and specify the access-to-subprogram type and the access-to-subprogram.

Selecting subprograms

A practical application of access to subprograms is that it enables us to dynamically select a subprogram and pass it to another subprogram, where it can then be called.

For example, we may have a `Process` procedure that receives a logging procedure as a parameter (`Log_Proc`). Also, this parameter may be `null` by default — so that no procedure is called if the parameter isn't specified:

Listing 1423: data_processing.ads

```

1  package Data_Processing is
2
3      type Data_Container is
4          array (Positive range <>) of Float;
5

```

(continues on next page)

(continued from previous page)

```

6  type Log_Procedure is
7      access procedure (D : Data_Container);
8
9  procedure Process
10     (D : in out Data_Container;
11      Log_Proc : Log_Procedure := null);
12
13 end Data_Processing;
```

Listing 1424: data_processing.adb

```

1  package body Data_Processing is
2
3      procedure Process
4          (D : in out Data_Container;
5           Log_Proc : Log_Procedure := null) is
6      begin
7          -- missing processing part...
8
9          if Log_Proc /= null then
10             Log_Proc (D);
11          end if;
12      end Process;
13
14 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳ Subprograms.Log_Procedure
 MD5: 59399e0809deb476f608faab7e4398bd

In the implementation of Process, we check whether Log_Proc is null or not. (If it's not null, we call the procedure. Otherwise, we just skip the call.)

Now, let's implement two logging procedures that match the expected form of the Log_Procedure type:

Listing 1425: log_element_per_line.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Data_Processing; use Data_Processing;
3
4  procedure Log_Element_Per_Line
5      (D : Data_Container) is
6  begin
7      Put_Line ("Elements: ");
8      for V of D loop
9          Put_Line (V'Image);
10      end loop;
11      Put_Line ("-----");
12 end Log_Element_Per_Line;
```

Listing 1426: log_csv.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Data_Processing; use Data_Processing;
3
4  procedure Log_Csv (D : Data_Container) is
5  begin
```

(continues on next page)

(continued from previous page)

```

6   for I in D'First .. D'Last - 1 loop
7       Put (D (I)'Image & ", ");
8   end loop;
9   Put (D (D'Last)'Image);
10  New_Line;
11 end Log_Csv;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Log_Procedure
 MD5: 468789f7331ffcd16f754f7116b076d7

Finally, we implement a test application that selects each of the logging procedures that we've just implemented:

Listing 1427: show_access_to_subprograms.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Data_Processing; use Data_Processing;
3
4  with Log_Element_Per_Line;
5  with Log_Csv;
6
7  procedure Show_Access_To_Subprograms is
8      D : Data_Container (1 .. 5) := (others => 1.0);
9  begin
10     Put_Line ("==== Log_Element_Per_Line ====");
11     Process (D, Log_Element_Per_Line'Access);
12
13     Put_Line ("==== Log_Csv ====");
14     Process (D, Log_Csv'Access);
15
16     Put_Line ("==== None ====");
17     Process (D);
18 end Show_Access_To_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Log_Procedure
 MD5: 134aa682cea1999efa0ea97052f315c8

Runtime output

```

==== Log_Element_Per_Line ====
Elements:
  1.00000E+00
  1.00000E+00
  1.00000E+00
  1.00000E+00
  1.00000E+00
-----
==== Log_Csv ====
  1.00000E+00, 1.00000E+00, 1.00000E+00, 1.00000E+00, 1.00000E+00
==== None ====

```

Here, we use the **Access** attribute to get access to the `Log_Element_Per_Line` and `Log_Csv` procedures. Also, in the third call, we don't pass any access as an argument, which is then **null** by default.

Null exclusion

We can use null exclusion when declaring an access to subprograms. By doing so, we ensure that a subprogram must be specified — either as a parameter or when initializing an access object. Otherwise, an exception is raised. Let's adapt the previous example and introduce the `Init_Function` type:

Listing 1428: data_processing.ads

```
1 package Data_Processing is
2
3     type Data_Container is
4         array (Positive range <>) of Float;
5
6     type Init_Function is
7         not null access function return Float;
8
9     procedure Process
10         (D : in out Data_Container;
11          Init_Func : Init_Function);
12
13 end Data_Processing;
```

Listing 1429: data_processing.adb

```
1 package body Data_Processing is
2
3     procedure Process
4         (D : in out Data_Container;
5          Init_Func : Init_Function) is
6     begin
7         for I in D'Range loop
8             D (I) := Init_Func.all;
9         end loop;
10    end Process;
11
12 end Data_Processing;
```

In this case, we specify that `Init_Function` is **not null access** because we want to always be able to call this function in the `Process` procedure (i.e. without raising an exception).

When an access to a subprogram doesn't have parameters — which is the case for the subprograms of `Init_Function` type — we need to explicitly dereference it by writing `.all`. (In this case, `.all` isn't optional.) Therefore, we have to write `Init_Func.all` in the implementation of the `Process` procedure of the code example.

Now, let's declare two simple functions — `Init_Zero` and `Init_One` — that return 0.0 and 1.0, respectively:

Listing 1430: init_zero.ads

```
1 function Init_Zero return Float;
```

Listing 1431: init_one.ads

```
1 function Init_One return Float;
```

Listing 1432: init_zero.adb

```
1 function Init_Zero return Float is
2 begin
3     return 0.0;
```

(continues on next page)

(continued from previous page)

```
4 end Init_Zero;
```

Listing 1433: init_one.adb

```
1 function Init_One return Float is
2 begin
3     return 1.0;
4 end Init_One;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳ Subprograms.Access_Init_Function
 MD5: 444110d50ddb430fd5be31cf1b417fc8

Finally, let's see a test application where we select each of the init functions we've just implemented:

Listing 1434: log_element_per_line.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 procedure Log_Element_Per_Line
5   (D : Data_Container) is
6 begin
7   Put_Line ("Elements: ");
8   for V of D loop
9     Put_Line (V'Image);
10  end loop;
11  Put_Line ("-----");
12 end Log_Element_Per_Line;
```

Listing 1435: show_access_to_subprograms.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Data_Processing; use Data_Processing;
3
4 with Init_Zero;
5 with Init_One;
6
7 with Log_Element_Per_Line;
8
9 procedure Show_Access_To_Subprograms is
10   D : Data_Container (1 .. 5) := (others => 1.0);
11 begin
12   Put_Line ("==== Init_Zero ====");
13   Process (D, Init_Zero'Access);
14   Log_Element_Per_Line (D);
15
16   Put_Line ("==== Init_One ====");
17   Process (D, Init_One'Access);
18   Log_Element_Per_Line (D);
19
20   -- Put_Line ("==== None ====");
21   -- Process (D, null);
22   -- Log_Element_Per_Line (D);
23 end Show_Access_To_Subprograms;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_Subprograms.Access_Init_Function
MD5: ae0e3fd58e9bb83061248967c709190a

Runtime output

```
==== Init_Zero ====
Elements:
 0.00000E+00
 0.00000E+00
 0.00000E+00
 0.00000E+00
 0.00000E+00
-----
==== Init_One ====
Elements:
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
 1.00000E+00
-----
```

Here, we use the **Access** attribute to get access to the `Init_Zero` and `Init_One` functions. Also, if we uncomment the call to `Process` with **null** as an argument for the init function, we see that the `Constraint_Error` exception is raised at run time — as the argument cannot be **null** due to the null exclusion.

For further reading...

Note

This example was originally written by Robert A. Duff and was part of the [Gem #24²⁷³](#).

Here's another example, first with **null**:

Listing 1436: `show_null_procedure.ads`

```
1 package Show_Null_Procedure is
2   type Element is limited null record;
3   -- Not implemented yet
4
5   type Ref_Element is access all Element;
6
7   type Table is limited null record;
8   -- Not implemented yet
9
10  type Iterate_Action is
11    access procedure
12      (X : not null Ref_Element);
13
14  procedure Iterate
15    (T      : Table;
16     Action : Iterate_Action := null);
17    -- If Action is null, do nothing.
18
19 end Show_Null_Procedure;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Null_Procedure
 MD5: ac21dd76ed9fb7f26839c24210cf4425

and without **null**:

Listing 1437: show_null_procedure.ads

```

1 package Show_Null_Procedure is
2   type Element is limited null record;
3   -- Not implemented yet
4
5   type Ref_Element is access all Element;
6
7   type Table is limited null record;
8   -- Not implemented yet
9
10  procedure Do_Nothing
11    (X : not null Ref_Element) is null;
12
13  type Iterate_Action is
14    access procedure
15      (X : not null Ref_Element);
16
17  procedure Iterate
18    (T      : Table;
19     Action : not null Iterate_Action
20      := Do_Nothing'Access);
21
22 end Show_Null_Procedure;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Null_Procedure
 MD5: 7341d8f23cd4efe45698481be452a9e8

The style of the second Iterate is clearly better because it makes use of the syntax to indicate that a procedure is expected. This is a complete package that includes both versions of the Iterate procedure:

Listing 1438: example.ads

```

1 package Example is
2
3   type Element is limited private;
4   type Ref_Element is access all Element;
5
6   type Table is limited private;
7
8   type Iterate_Action is
9     access procedure
10      (X : not null Ref_Element);
11
12  procedure Iterate
13    (T : Table;
14     Action : Iterate_Action := null);
15    -- If Action is null, do nothing.
16
17  procedure Do_Nothing
18    (X : not null Ref_Element) is null;
19  procedure Iterate_2
20    (T : Table;
21     Action : not null Iterate_Action
```



```

22         := Do_Nothing'Access);
23
24 private
25     type Element is limited
26         record
27             Component : Integer;
28         end record;
29     type Table is limited null record;
30 end Example;

```

Listing 1439: example.adb

```

1 package body Example is
2
3     An_Element : aliased Element;
4
5     procedure Iterate
6         (T : Table;
7          Action : Iterate_Action := null)
8     is
9     begin
10         if Action /= null then
11             Action (An_Element'Access);
12             -- In a real program, this would do
13             -- something more sensible.
14         end if;
15     end Iterate;
16
17     procedure Iterate_2
18         (T : Table;
19          Action : not null Iterate_Action
20              := Do_Nothing'Access)
21     is
22     begin
23         Action (An_Element'Access);
24         -- In a real program, this would do
25         -- something more sensible.
26     end Iterate_2;
27
28 end Example;

```

Listing 1440: show_example.adb

```

1 with Example; use Example;
2
3 procedure Show_Example is
4     T : Table;
5 begin
6     Iterate_2 (T);
7 end Show_Example;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 Subprograms.Complete_Not_Null_Procedure
 MD5: ab0a41e0d39a8a16b0b69f8c6b2a43fd

Writing **not null** Iterate_Action might look a bit more complicated, but it's worthwhile, and anyway, as mentioned earlier, the compatibility requirement requires that the **not null** be explicit, rather than the other way around.

²⁷³ <https://www.adacore.com/gems/ada-gem-24>

Access to protected subprograms

Up to this point, we've discussed access to *normal* Ada subprograms. In some situations, however, we might want to have access to protected subprograms. To do this, we can simply declare a type using **access protected**:

Listing 1441: simple_protected_access.ads

```

1 package Simple_Protected_Access is
2
3     type Access_Proc is
4       access protected procedure;
5
6     protected Obj is
7
8       procedure Do_Something;
9
10    end Obj;
11
12    Acc : Access_Proc := Obj.Do_Something'Access;
13
14 end Simple_Protected_Access;
```

Listing 1442: simple_protected_access.adb

```

1 package body Simple_Protected_Access is
2
3     protected body Obj is
4
5       procedure Do_Something is
6       begin
7         -- Not doing anything
8         -- for the moment...
9         null;
10      end Do_Something;
11
12    end Obj;
13
14 end Simple_Protected_Access;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳Subprograms.Simple_Protected_Access
 MD5: d82f7c90355e9810bd1e35f65e278626

Here, we declare the `Access_Proc` type as an access type to protected procedures. Then, we declare the variable `Acc` and assign to it the access to the `Do_Something` procedure (of the protected object `Obj`).

Now, let's discuss a more useful example: a simple system that allows us to register protected procedures and execute them. This is implemented in `Work_Registry` package:

Listing 1443: work_registry.ads

```

1 package Work_Registry is
2
3     type Work_Id is tagged limited private;
4
5     type Work_Handler is
6       access protected procedure (T : Work_Id);
7
```

(continues on next page)

(continued from previous page)

```

8  subtype Valid_Work_Handler is
9      not null Work_Handler;
10
11  type Work_Handlers is
12      array (Positive range <>) of Work_Handler;
13
14  protected type Work_Handler_Registry
15      (Last : Positive)
16  is
17
18      procedure Register (T : Valid_Work_Handler);
19
20      procedure Reset;
21
22      procedure Process_All;
23
24  private
25
26      D      : Work_Handlers (1 .. Last);
27      Curr : Natural := 0;
28
29  end Work_Handler_Registry;
30
31  private
32
33      type Work_Id is tagged limited null record;
34
35  end Work_Registry;

```

Listing 1444: work_registry.adb

```

1  package body Work_Registry is
2
3      protected body Work_Handler_Registry is
4
5          procedure Register (T : Valid_Work_Handler)
6          is
7              begin
8                  if Curr < Last then
9                      Curr := Curr + 1;
10                     D (Curr) := T;
11                 end if;
12             end Register;
13
14             procedure Reset is
15                 begin
16                     Curr := 0;
17                 end Reset;
18
19             procedure Process_All is
20                 Dummy_ID : Work_Id;
21                 begin
22                     for I in D'First .. Curr loop
23                         D (I).all (Dummy_ID);
24                     end loop;
25                 end Process_All;
26
27             end Work_Handler_Registry;
28
29  end Work_Registry;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↳ Subprograms.Protected_Access_Init_Function
 MD5: 5dfa8ab098900ab4f6b7575e1cde5e53

Here, we declare the protected `Work_Handler_Registry` type with the following subprograms:

- `Register`, which we can use to register a protected procedure;
- `Reset`, which we can use to reset the system; and
- `Process_All`, which we can use to call all procedures that were registered in the system.

`Work_Handler` is our access to protected subprogram type. Also, we declare the `Valid_Work_Handler` subtype, which excludes `null`. By doing so, we can ensure that only valid procedures are passed to the `Register` procedure. In the protected `Work_Handler_Registry` type, we store the procedures in an array (of `Work_Handlers` type).

Important

Note that, in the type declaration `Work_Handler`, we say that the protected procedure must have a parameter of `Work_Id` type. In this example, this parameter is just used to *bind* the procedure to the `Work_Handler_Registry` type. The `Work_Id` type itself is actually declared as a null record (in the private part of the package), and it isn't really useful on its own.

If we had declared `type Work_Handler is access protected procedure;` instead, we would be able to register *any* protected procedure into the system, even the ones that might not be suitable for the system. By using a parameter of `Work_Id` type, however, we make use of strong typing to ensure that only procedures that were designed for the system can be registered.

In the next part of the code, we declare the `Integer_Storage` type, which is a simple protected type that we use to store an integer value:

Listing 1445: `integer_storage_system.ads`

```

1  with Work_Registry;
2
3  package Integer_Storage_System is
4
5      protected type Integer_Storage is
6
7          procedure Set (V : Integer);
8
9          procedure Show (T : Work_Registry.Work_Id);
10
11     private
12
13         I : Integer := 0;
14
15     end Integer_Storage;
16
17     type Integer_Storage_Access is
18         access Integer_Storage;
19
20     type Integer_Storage_Array is

```

(continues on next page)

(continued from previous page)

```

21     array (Positive range <>) of
22         Integer_Storage_Access;
23
24 end Integer_Storage_System;
```

Listing 1446: integer_storage_system.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Integer_Storage_System is
4
5      protected body Integer_Storage is
6
7          procedure Set (V : Integer) is
8              begin
9                  I := V;
10             end Set;
11
12             procedure Show (T : Work_Registry.Work_Id)
13                 is
14                     pragma Unreferenced (T);
15                 begin
16                     Put_Line ("Value: " & Integer'Image (I));
17                 end Show;
18
19         end Integer_Storage;
20
21     end Integer_Storage_System;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪ Subprograms.Protected_Access_Init_Function
 MD5: a388d792bc85709785d324c914d9d236

For the Integer_Storage type, we declare two procedures:

- Set, which we use to assign a value to the (protected) integer value; and
- Show, which we use to show the integer value that is stored in the protected object.

The Show procedure has a parameter of Work_Id type, which indicates that this procedure was designed to be registered in the system of Work_Handler_Registry type.

Finally, we have a test application in which we declare a registry (WHR) and an array of "protected integer objects" (Int_Stor):

Listing 1447: show_access_to_protected_subprograms.adb

```

1  with Work_Registry;
2  use Work_Registry;
3
4  with Integer_Storage_System;
5  use Integer_Storage_System;
6
7  procedure Show_Access_To_Protected_Subprograms is
8
9      WHR      : Work_Handler_Registry (5);
10     Int_Stor : Integer_Storage_Array (1 .. 3);
11
12     begin
13         -- Allocate and initialize integer storage
```

(continues on next page)

(continued from previous page)

```

14  --
15  -- (For the initialization, we're just
16  -- assigning the index here, but we could
17  -- really have used any integer value.)
18
19  for I in Int_Stor'Range loop
20      Int_Stor (I) := new Integer_Storage;
21      Int_Stor (I).Set (I);
22  end loop;
23
24  -- Register handlers
25
26  for I in Int_Stor'Range loop
27      WHR.Register (Int_Stor (I).all.Show'Access);
28  end loop;
29
30  -- Now, use Process_All to call the handlers
31  -- (in this case, the Show procedure for
32  -- each protected object from Int_Stor).
33
34  WHR.Process_All;
35
36  end Show_Access_To_Protected_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Protected_Access_Init_Function
 MD5: 44c24ef07333e1d31844cc2ea6d91ab6

Runtime output

```

Value: 1
Value: 2
Value: 3

```

The work handler registry (WHR) has a maximum capacity of five procedures, whereas the Int_Stor array has a capacity of three elements. By calling WHR.Register and passing Int_Stor (I).all.Show'Access, we register the Show procedure of each protected object from Int_Stor.

Important

Note that the components of the Int_Stor array are of Integer_Storage_Access type, which is declared as an access to Integer_Storage objects. Therefore, we have to dereference the object (by writing Int_Stor (I).all) before getting access to the Show procedure (by writing .Show'Access).

We have to use an access type here because we cannot pass the access (to the Show procedure) of a local object in the call to the Register procedure. Therefore, the protected objects (of Integer_Storage type) cannot be local.

This issue becomes evident if we replace the declaration of Int_Stor with a local array (and then adapt the remaining code). If we do this, we get a compilation error in the call to Register:

Listing 1448: show_access_to_protected_subprograms.adb

```

1  with Work_Registry;
2  use  Work_Registry;
3
4  with Integer_Storage_System;
5  use  Integer_Storage_System;
6
7  procedure Show_Access_To_Protected_Subprograms
8  is
9      WHR      : Work_Handler_Registry (5);
10
11     Int_Stor : array (1 .. 3) of Integer_Storage;
12
13  begin
14      -- Allocate and initialize integer storage
15      --
16      -- (For the initialization, we're just
17      -- assigning the index here, but we could
18      -- really have used any integer value.)
19
20     for I in Int_Stor'Range loop
21         -- Int_Stor (I) := new Integer_Storage;
22         Int_Stor (I).Set (I);
23     end loop;
24
25     -- Register handlers
26
27     for I in Int_Stor'Range loop
28         WHR.Register (Int_Stor (I).Show'Access);
29         -- ^ ERROR!
30     end loop;
31
32     -- Now, call the handlers
33     -- (i.e. the Show procedure of each
34     -- protected object).
35
36     WHR.Process_All;
37
38 end Show_Access_To_Protected_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_To_
 ↪Subprograms.Protected_Access_Init_Function
 MD5: 359241c84cd30313fe2d7701b55f303e

Build output

```

show_access_to_protected_subprograms.adb:28:21: error: non-local pointer cannot
↪point to local object
gprbuild: *** compilation phase failed

```

As we've just discussed, this error is due to the fact that `Int_Stor` is now a "local" protected object, and the accessibility rules don't allow mixing it with non-local accesses in order to prevent the possibility of dangling references.

When we call `WHR.Process_All`, the registry system calls each procedure that has been registered with the system. When looking at the values displayed by the test application, we may notice that each call to `Show` is referring to a different protected object. In fact, even though we're passing just the access to a protected *procedure* in the call to `Register`, that access is also associated to a specific protected object. (This is different from access to non-protected subprograms we've discussed previously: in that case, there's no object

associated.) If we replace the argument to Register by `Int_Stor (2).all.Show'Access`, for example, the three Show procedures registered in the system will now refer to the same protected object (stored at `Int_Stor (2)`).

Also, even though we have registered the same procedure (Show) of the same type (Integer_Storage) in all calls to Register, we could have used a different protected procedure — and of a different protected type. As an exercise, we could, for example, create a new type called Float_Storage (based on the code that we used for the Integer_Storage type) and register some objects of Float_Storage type into the system (with a couple of additional calls to Register). If we then call `WHR.Process_All`, we'd see that the system is able to cope with objects of both Integer_Storage and Float_Storage types. In fact, the system implemented with the Work_Handler_Registry can be seen as "type agnostic," as it doesn't care about which type the protected objects have — as long as the subprograms we want to register are conformant to the Valid_Work_Handler type.

28.1.16 Accessibility Rules and Access-To-Subprograms

In general, the accessibility rules that we discussed *previously for access-to-objects* (page 932) also apply to access-to-subprograms. In this section, we discuss minor differences when applying those rules to access-to-subprograms.

In our discussion about accessibility rules, we've looked into *accessibility levels* (page 934) and the *accessibility rules* (page 934) that are based on those levels. The same accessibility rules apply to access-to-subprograms. *As we said previously* (page 937), operations targeting objects at a *less-deep* level are illegal, as it's the case for subprograms as well:

Listing 1449: access_to_subprogram_types.ads

```

1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
5
6     type Access_To_Function is
7         access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;
```

Listing 1450: show_access_to_subprogram_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7     Func : Access_To_Function;
8
9     Value : Integer := 0;
10 begin
11     declare
12         function Add_One (I : Integer)
13             return Integer is
14             (I + 1);
15     begin
16         Func := Add_One'Access;
17         -- This assignment is illegal because the
18         -- Access_To_Function type is less deep
19         -- than Add_One.
20     end;
21
```

(continues on next page)

(continued from previous page)

```
22   Put_Line ("Value: " & Value'Image);
23   Value := Func (Value);
24   Put_Line ("Value: " & Value'Image);
25 end Show_Access_To_Subprogram_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Less_Deep
MD5: 2a068732606a1fee156e82515febe9c4

Build output

```
show_access_to_subprogram_error.adb:16:15: error: subprogram must not be deeper_
↳ than access type
gprbuild: *** compilation phase failed
```

Obviously, we can correct this error by putting the Add_One function at the same level as the Access_To_Function type, i.e. at library level:

Listing 1451: access_to_subprogram_types.ads

```
1 package Access_To_Subprogram_Types is
2
3   type Access_To_Procedure is
4     access procedure (I : in out Integer);
5
6   type Access_To_Function is
7     access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;
```

Listing 1452: add_one.ads

```
1 function Add_One (I : Integer) return Integer;
```

Listing 1453: add_one.adb

```
1 function Add_One (I : Integer) return Integer is
2 begin
3   return I + 1;
4 end Add_One;
```

Listing 1454: show_access_to_subprogram_error.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 with Add_One;
7
8 procedure Show_Access_To_Subprogram_Error is
9   Func : Access_To_Function;
10
11   Value : Integer := 0;
12 begin
13   Func := Add_One'Access;
14
15   Put_Line ("Value: " & Value'Image);
```

(continues on next page)

(continued from previous page)

```

16     Value := Func (Value);
17     Put_Line ("Value: " & Value'Image);
18 end Show_Access_To_Subprogram_Error;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_
 ↳ Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Less_Deep_Fix
 MD5: 7f7488c541fb457ced653a2e6cc2fad1

Runtime output

```

Value:  0
Value:  1

```

As a recommendation, resolving accessibility issues in the case of access-to-subprograms is best done by refactoring the subprograms of your source code — for example, moving subprograms to a different level.

Unchecked Access

Previously, we discussed about the *Unchecked_Access attribute* (page 942), which we can use to circumvent accessibility issues in specific cases for access-to-objects. We also said in that section that this attribute only exists for objects, not for subprograms. We can use the previous example to illustrate this limitation:

Listing 1455: access_to_subprogram_types.ads

```

1 package Access_To_Subprogram_Types is
2
3     type Access_To_Procedure is
4         access procedure (I : in out Integer);
5
6     type Access_To_Function is
7         access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;

```

Listing 1456: show_access_to_subprogram_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use   Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7     Func : Access_To_Function;
8
9     function Add_One (I : Integer)
10         return Integer is
11         (I + 1);
12
13     Value : Integer := 0;
14 begin
15     Func := Add_One'Access;
16
17     Put_Line ("Value: " & Value'Image);
18     Value := Func (Value);
19     Put_Line ("Value: " & Value'Image);
20 end Show_Access_To_Subprogram_Error;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Rules_
↳ Access_To_Subprograms.Access_To_Subprogram_Accessibility_Error_Same_Lifetime
MD5: c1ee1946f0c979eb30fbf2c72c426f50
```

Build output

```
show_access_to_subprogram_error.adb:15:12: error: subprogram must not be deeper_
↳ than access type
gprbuild: *** compilation phase failed
```

When we analyze the `Show_Access_To_Subprogram_Error` procedure, we see that the `Func` object and the `Add_One` function have the same lifetime. Therefore, in this very specific case, we could safely assign `Add_One`' `Access` to `Func` and call `Func` for `Value`. Due to the accessibility rules, however, this assignment is illegal. (Obviously, the accessibility issue here is that the `Access_To_Function` type has a potentially longer lifetime.)

In the case of access-to-objects, we could use `Unchecked_Access` to enforce assignments that we consider safe after careful analysis. However, because this attribute isn't available for access-to-subprograms, the best solution is to move the subprogram to a level that allows the assignment to be legal, as we said before.

In the GNAT toolchain

GNAT offers an equivalent for `Unchecked_Access` that can be used for subprograms: the `Unrestricted_Access` attribute. Note, however, that this attribute is not portable.

Listing 1457: `access_to_subprogram_types.ads`

```
1 package Access_To_Subprogram_Types is
2
3   type Access_To_Procedure is
4     access procedure (I : in out Integer);
5
6   type Access_To_Function is
7     access function (I : Integer) return Integer;
8
9 end Access_To_Subprogram_Types;
```

Listing 1458: show_access_to_subprogram_error.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7     Func : Access_To_Function;
8
9     function Add_One (I : Integer)
10         return Integer is
11         (I + 1);
12
13     Value : Integer := 0;
14 begin
15     Func := Add_One'Unrestricted_Access;
16     --      ~~~~~
17     --      Allowing access to local function
18
19     Put_Line ("Value: " & Value'Image);
20     Value := Func (Value);
21     Put_Line ("Value: " & Value'Image);
22 end Show_Access_To_Subprogram_Error;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_
 Rules_Access_To_Subprograms.Unrestricted_Access
 MD5: 90e2c57c01463cbe6efee6e093d01e5b

Runtime output

Value: 0
 Value: 1

As we can see, the `Unrestricted_Access` attribute can be safely used in this specific case to circumvent the accessibility rule limitation.

28.1.17 Access and Address

As we know, an access type is not a pointer, and it doesn't just indicate an address in memory. In fact, to represent an address in Ada, we use *the Address type* (page 397). Also, as we discussed earlier, we can use operators such as `<`, `>`, `+` and `-` for addresses. In contrast to that, those operators aren't available for access types — except, of course, for `=` and `/=`.

In certain situations, however, we might need to convert between access types and addresses. In this section, we discuss how to do so.

i In the Ada Reference Manual

- 13.3 Operational and Representation Attributes²⁷⁴
- 13.7 The Package System²⁷⁵

²⁷⁴ <http://www.ada-auth.org/standards/22rm/html/RM-13-3.html>

²⁷⁵ <http://www.ada-auth.org/standards/22rm/html/RM-13-7.html>

Address and access conversion

The generic `System.Address_To_Access_Conversions` package allows us to convert between access types and addresses. This might be useful for specific low-level operations. Let's see an example:

Listing 1459: `show_address_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with System.Address_To_Access_Conversions;
4  with System.Address_Image;
5
6  procedure Show_Address_Conversion is
7
8      package Integer_AAC is
9          new System.Address_To_Access_Conversions
10             (Object => Integer);
11      use Integer_AAC;
12
13      subtype Integer_Access is
14          Integer_AAC.Object_Pointer;
15      -- This is similar to:
16      --
17      -- type Integer_Access is access all Integer;
18
19      I : aliased Integer := 5;
20      AI : Integer_Access := I'Access;
21  begin
22      Put_Line ("I'Address : "
23              & System.Address_Image (I'Address));
24
25      Put_Line ("AI.all'Address : "
26              & System.Address_Image
27              (AI.all'Address));
28
29      Put_Line ("To_Address (AI) : "
30              & System.Address_Image
31              (To_Address (AI)));
32  end Show_Address_Conversion;

```

Code block metadata

Project: `Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.Address_Conversion`
MD5: 717532026247044a667b60f6c1e1c7da

Runtime output

```

I'Address : 00007FFF9FB7E804
AI.all'Address : 00007FFF9FB7E804
To_Address (AI) : 00007FFF9FB7E804

```

In this example, we instantiate the generic `System.Address_To_Access_Conversions` package using `Integer` as our target object type. This new package (`Integer_AAC`) has an `Object_Pointer` type, which is equivalent to a declaration such as `type Integer_Access is access all Integer`. (In this example, we declare `Integer_Access` as a subtype of `Integer_AAC.Object_Pointer` to illustrate that.)

The `Integer_AAC` package also includes the `To_Address` function, which converts an access object to an address. If the actual parameter is not null, `To_Address` returns the same information as if we were using the `Address` attribute for the designated object. In other words, `To_Address (AI) = AI.all'Address` when `AI /= null`.

If the access value is null, `To_Address` returns `Null_Address`, while `.all'Address` makes the *access check* (page 808) fail because we have to dereference the access object (via `.all`) before retrieving its address (via the `Address` attribute).

In addition to the `To_Address` function, the `To_Pointer` function is available to convert from an address to an object of access type. For example:

Listing 1460: `show_address_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with System;      use System;
3
4  with System.Address_To_Access_Conversions;
5  with System.Address_Image;
6
7  procedure Show_Address_Conversion is
8
9      package Integer_AAC is
10         new System.Address_To_Access_Conversions
11         (Object => Integer);
12     use Integer_AAC;
13
14     subtype Integer_Access is
15         Integer_AAC.Object_Pointer;
16
17     I      : aliased Integer := 5;
18     AI_1, AI_2 : Integer_Access;
19     A      : Address;
20 begin
21     AI_1 := I'Access;
22     A    := To_Address (AI_1);
23     AI_2 := To_Pointer (A);
24
25     Put_Line ("AI_1.all'Address : "
26             & System.Address_Image
27             (AI_1.all'Address));
28     Put_Line ("AI_2.all'Address : "
29             & System.Address_Image
30             (AI_2.all'Address));
31
32     if AI_1 = AI_2 then
33         Put_Line ("AI_1 = AI_2");
34     else
35         Put_Line ("AI_1 /= AI_2");
36     end if;
37 end Show_Address_Conversion;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.
 ↪Address_Conversion
 MD5: 5c6fc19ca1aa227feba97ea610dd9218

Runtime output

```

AI_1.all'Address : 00007FFEAA60888C
AI_2.all'Address : 00007FFEAA60888C
AI_1 = AI_2

```

Here, we convert the `A` address back to an access value by calling `To_Pointer (A)`. (When running this object, we see that `AI_1` and `AI_2` have the same access value.)

Conversion of unbounded designated types

Note that the conversions might not work in all cases. For instance, when the designated type — indicated by the formal `Object` parameter of the generic `Address_To_Access_Conversions` package — is unbounded, the result of a call to `To_Pointer` may not have bounds.

Let's adapt the previous code example and replace the **Integer** type by the (unbounded) **String** type:

Listing 1461: `show_address_conversion.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with System;      use System;
3
4  with System.Address_To_Access_Conversions;
5  with System.Address_Image;
6
7  procedure Show_Address_Conversion is
8
9      package String_AAC is
10         new System.Address_To_Access_Conversions
11             (Object => String);
12         use String_AAC;
13
14         subtype Integer_Access is
15             String_AAC.Object_Pointer;
16
17         S          : aliased String := "Hello";
18         AI_1, AI_2 : Integer_Access;
19         A          : Address;
20     begin
21         AI_1 := S'Access;
22         A    := To_Address (AI_1);
23
24         AI_2 := To_Pointer (A);
25         -- ^^^^^^^^^^^^^^
26         -- WARNING: Result might not have bounds
27
28         Put_Line ("AI_1.all'Address : "
29                 & System.Address_Image
30                     (AI_1.all'Address));
31         Put_Line ("AI_2.all'Address : "
32                 & System.Address_Image
33                     (AI_2.all'Address));
34
35         if AI_1 = AI_2 then
36             Put_Line ("AI_1 = AI_2");
37         else
38             Put_Line ("AI_1 /= AI_2");
39         end if;
40
41         Put_Line ("AI_1: " & AI_1.all);
42         Put_Line ("AI_2: " & AI_2.all);
43         -- ^^^^^^^^^^
44         -- WARNING: As AI_2 might not have bounds
45         --             due to the call to To_Pointer
46         --             the behavior of this call to
47         --             the "&" operator is
48         --             unpredictable.
49     end Show_Address_Conversion;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Access_Address.
↳Address_Conversion
MD5: b1adcaa1f2cb4dfbd157aebf7893bd72
```

Build output

```
show_address_conversion.adb:9:04: warning: in instantiation at s-atacco.ads:46
↳[enabled by default]
show_address_conversion.adb:9:04: warning: Object is unconstrained array type
↳[enabled by default]
show_address_conversion.adb:9:04: warning: To_Pointer results may not have bounds
↳[enabled by default]
```

Runtime output

```
AI_1.all'Address : 00007FFDB8E98D88
AI_2.all'Address : 00007FFDB8E98D88
AI_1 = AI_2
AI_1: Hello
AI_2: Hello
```

In this case, the call to `To_Pointer` (A) might not have bounds, so any operation on `AI_2` might lead to unpredictable results.

In the Ada Reference Manual

- [13.7.2 The Package System.Address_To_Access_Conversions²⁷⁶](#)

28.2 Anonymous Access Types

28.2.1 Named and Anonymous Access Types

The previous chapter dealt with access type declarations such as this one:

```
type Integer_Access is access all Integer;

procedure Add_One (A : Integer_Access);
```

In addition to named access type declarations such as the one in this example, Ada also supports anonymous access types, which, as the name implies, don't have an actual type declaration.

To declare an access object of anonymous type, we just specify the subtype of the object or subprogram we want to have access to. For example:

```
procedure Add_One (A : access Integer);
```

When we compare this example with the previous one, we see that the declaration `A : Integer_Access` becomes `A : access Integer`. Here, `access Integer` is the anonymous access type declaration, and `A` is an access object of this anonymous type.

To be more precise, `A : access Integer` is an *access parameter* (page 1021) and it's specifying an *anonymous access-to-object type* (page 1002). Another flavor of anonymous access types are *anonymous access-to-subprograms* (page 1044). We discuss all these topics in more details later.

Let's see a complete example:

²⁷⁶ <http://www.ada-auth.org/standards/22rm/html/RM-13-7-2.html>

Listing 1462: show_anonymous_access_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Anonymous_Access_Types is
4   I_Var : aliased Integer;
5
6   A      : access Integer;
7   --      ^ Anonymous access type
8 begin
9   A := I_Var'Access;
10  --      ^ Assignment to object of
11  --      anonymous access type.
12
13  A.all := 22;
14
15  Put_Line ("A.all: " & Integer'Image (A.all));
16 end Show_Anonymous_Access_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_Types.Simple_Anonymous_Access_Types
 MD5: f0c92c76d970089c1d503c599d6869dd

Runtime output

A.all: 22

Here, A is an access object whose value is initialized with the access to I_Var. Because the declaration of A includes the declaration of an anonymous access type, we don't declare an extra Integer_Access type, as we did in previous code examples.

In the Ada Reference Manual

- [3.10 Access Types](#)²⁷⁷

Relation to named types

Anonymous access types were not part of the first version of the Ada standard, which only had support for named access types. They were introduced later to cover some use-cases that were difficult — or even impossible — with access types.

In this sense, anonymous access types aren't just access types without names. Certain accessibility rules for anonymous access types are a bit less strict. In those cases, it might be interesting to consider using them instead of named access types.

In general, however, we should only use anonymous access types in those specific cases where using named access types becomes too cumbersome. As a general recommendation, we should give preference to named access types whenever possible. (Anonymous access-to-object types have *drawbacks that we discuss later* (page 1004).)

Benefits of anonymous access types

One of the main benefits of anonymous access types is their flexibility: since there isn't an explicit access type declaration associated with them, we only have to worry about the subtype S we intend to access.

²⁷⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Also, as long as the subtype `S` in a declaration `access S` is always the same, no conversion is needed between two access objects of that anonymous type, and the `S'Access` attribute always works.

Let's see an example:

Listing 1463: show.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show (Name : String;
4                V    : access Integer) is
5 begin
6   Put_Line (Name & ".all: "
7             & Integer'Image (V.all));
8 end Show;
```

Listing 1464: show_anonymous_access_types.adb

```

1 with Show;
2
3 procedure Show_Anonymous_Access_Types is
4   I_Var : aliased Integer;
5   A     : access Integer;
6   B     : access Integer;
7 begin
8   A := I_Var'Access;
9   B := A;
10
11   A.all := 22;
12
13   Show ("A", A);
14   Show ("B", B);
15 end Show_Anonymous_Access_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_Types.Anonymous_Access_Object_Assignment
 MD5: 2822ca0bd6ac251dccc1ced60747fbe1

Runtime output

```

A.all:  22
B.all:  22
```

In this example, we have two access objects `A` and `B`. Since they're objects of anonymous access types that refer to the same subtype `Integer`, we can assign `A` to `B` without a type conversion, and pass those access objects as an argument to the `Show` procedure.

(Note that the use of an access parameter in the `Show` procedure is for demonstration purpose only: a simply `Integer` as the type of this input parameter would have been more than sufficient to implement the procedure. Actually, in this case, avoiding the access parameter would be the recommended approach in terms of clean Ada software design.)

In contrast, if we had used named type declarations, the code would be more complicated and more limited:

Listing 1465: aux.ads

```

1 package Aux is
2
```

(continues on next page)

(continued from previous page)

```
3  type Integer_Access is access all Integer;
4
5  procedure Show (Name : String;
6                 V     : Integer_Access);
7
8  end Aux;
```

Listing 1466: aux.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Aux is
4
5      procedure Show (Name : String;
6                     V     : Integer_Access) is
7      begin
8          Put_Line (Name & ".all: "
9                  & Integer'Image (V.all));
10     end Show;
11
12 end Aux;
```

Listing 1467: show_anonymous_access_types.adb

```
1  with Aux; use Aux;
2
3  procedure Show_Anonymous_Access_Types is
4      -- I_Var : aliased Integer;
5
6      A : Integer_Access;
7      B : Integer_Access;
8  begin
9      -- A := I_Var'Access;
10     --     ^ ERROR: non-local pointer cannot
11     --         point to local object.
12
13     A := new Integer;
14     B := A;
15
16     A.all := 22;
17
18     Show ("A", A);
19     Show ("B", B);
20 end Show_Anonymous_Access_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_Types.Anonymous_Access_Object_Assignment
MD5: 681c2cf7f5e8d520490cc5594484ce69

Runtime output

```
A.all: 22
B.all: 22
```

Here, apart from the access type declaration (`Integer_Access`), we had to make two adaptations to convert the previous code example:

1. We had to move the `Show` procedure to a package (which we simply called `Aux`) because of the access type declaration.

- Also, we had to allocate an object for A instead of retrieving the access attribute of I_Var because we cannot use a pointer to a local object in the assignment to a non-local pointer, as indicate in the comments.

This restriction regarding non-local pointer assignments is an example of the stricter accessibility rules that apply to named access types. As mentioned earlier, the S'Access attribute always works when we use anonymous access types — this is not always the case for named access types.

Important

As mentioned earlier, if we want to use two access objects in an operation, the rule says that the subtype S of the anonymous type used in their corresponding declaration must match. In the following example, we can see how this rule works:

Listing 1468: show_anonymous_access_subtype_error.adb

```

1  procedure Show_Anonymous_Access_Subtype_Error is
2      subtype Integer_1_10 is Integer range 1 .. 10;
3
4      I_Var : aliased Integer;
5      A     : access Integer := I_Var'Access;
6      B     : access Integer_1_10;
7  begin
8      A := I_Var'Access;
9
10     B := A;
11     -- ^ ERROR: subtype doesn't match!
12
13     B := I_Var'Access;
14     -- ^ ERROR: subtype doesn't match!
15 end Show_Anonymous_Access_Subtype_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_Types.Anonymous_Access_Subtype_Error
MD5: cecfe703ea8b42bad61c45f33cbcb67b

Build output

```

show_anonymous_access_subtype_error.adb:10:09: error: target_
↳ designated subtype not compatible with type "Standard.Integer"
show_anonymous_access_subtype_error.adb:13:09: error: object subtype_
↳ must statically match designated subtype
gprbuild: *** compilation phase failed
```

Even though Integer_1_10 is a subtype of **Integer**, we cannot assign A to B because the subtype that their access type declarations refer to — **Integer** and Integer_1_10, respectively — doesn't match. The same issue occurs when retrieving the access attribute of I_Var in the assignment to B.

The later sections on [anonymous access-to-object type](#) (page 1002) and [anonymous access-to-subprograms](#) (page 1044) cover more specific details on anonymous access types.

28.2.2 Anonymous Access-To-Object Types

In the *previous chapter* (page 881), we introduced named access-to-object types and used those types throughout the chapter. Also, in the *previous section* (page 997), we've seen some simple examples of anonymous access-to-object types:

```
procedure Add_One (A : access Integer);
--                ^ Anonymous access type

A : access Integer;
-- ^ Anonymous access type
```

In addition to parameters and objects, we can use anonymous access types in discriminants, components of array and record types, renamings and function return types. (We discuss *anonymous access discriminants* (page 1011) and *anonymous access parameters* (page 1021) later on.) Let's see a code example that includes all these cases:

Listing 1469: all_anonymous_access_to_object_types.ads

```
1 package All_Anonymous_Access_To_Object_Types is
2
3   procedure Add_One (A : access Integer) is null;
4   --                ^ Anonymous access type
5
6   AI : access Integer;
7   -- ^ Anonymous access type
8
9   type Rec (AI : access Integer) is private;
10  -- ^ Anonymous access type
11
12  type Access_Array is
13    array (Positive range <>) of
14      access Integer;
15  -- ^ Anonymous access type
16
17  Arr : array (1 .. 5) of access Integer;
18  -- ^ Anonymous access type
19
20  AI_Renaming : access Integer renames AI;
21  -- ^ Anonymous access type
22
23  function Init_Access_Integer
24    return access Integer is (null);
25  -- ^ Anonymous access type
26
27 private
28
29   type Rec (AI : access Integer) is record
30   -- ^ Anonymous access type
31     Internal_AI : access Integer;
32   -- ^ Anonymous access type
33
34   end record;
35
36 end All_Anonymous_Access_To_Object_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.All_Anonymous_Access_To_Object_Types
MD5: 6533b22a4e4526702320cb327bf6f69a

In this example, we see multiple examples of anonymous access-to-object types:

- as the A parameter of the Add_One procedure;
- in the declaration of the AI access object;
- as the AI discriminant of the Rec type;
- as the component type of the Access_Array type;
- as the component type of the Arr array;
- in the AI_Renaming renaming;
- as the return type of the Init_Access_Integer;
- as the Internal_AI of component of the Rec type.

i In the Ada Reference Manual

- 3.10 Access Types²⁷⁸

Not Null Anonymous Access-To-Object Types

As expected, **null** is a valid value for an anonymous access type. However, we can forbid **null** as a valid value by using **not null** in the anonymous access type declaration. For example:

Listing 1470: all_anonymous_access_to_object_types.ads

```

1 package All_Anonymous_Access_To_Object_Types is
2
3   procedure Add_One (A : not null access Integer)
4     is null;
5   -- ^ Anonymous access type
6
7   I : aliased Integer;
8
9   AI : not null access Integer := I'Access;
10  -- ^ Anonymous access type
11  -- ^^^^^^^
12  -- Initialization required!
13
14  type Rec (AI : not null access Integer) is
15    private;
16  -- ^ Anonymous access type
17
18  type Access_Array is
19    array (Positive range <>) of
20      not null access Integer;
21  -- ^ Anonymous access type
22
23  Arr : array (1 .. 5) of
24    not null access Integer :=
25  -- ^ Anonymous access type
26    (others => I'Access);
27  -- ^^^^^^^^^^^^^^^^^
28  -- Initialization required!
29
30  AI_Renaming : not null access Integer
31    renames AI;
32  -- ^ Anonymous access type
33
```

(continues on next page)

²⁷⁸ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

34  function Init_Access_Integer
35      return not null access Integer is (I'Access);
36      --      ^ Anonymous access type
37      --      ^^^^^^^
38      --      Initialization required!
39
40  private
41
42      type Rec (AI : not null access Integer) is
43          record
44              --      ^ Anonymous access type
45              Internal_AI : not null access Integer;
46              --      ^ Anonymous access type
47
48          end record;
49
50  end All_Anonymous_Access_To_Object_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.All_Not_Null_Anonymous_Access_To_Object_Types
 MD5: 027430aa9d5e19979206110f5e260d13

As you might have noticed, we took the previous code example and used **not null** for each usage instance of the anonymous access type. In this sense, this version of the code example is very similar to the previous one. Note, however, that we now have to explicitly initialize some elements to avoid the `Constraint_Error` exception being raised at runtime. This is the case for example for the AI access object:

```
AI : not null access Integer := I'Access;
```

If we hadn't initialized AI explicitly with `I'Access`, it would have been set to **null**, which would fail the **not null** constraint of the anonymous access type. Similarly, we also have to initialize the Arr array and return a valid access object for the `Init_Access_Integer` function.

Drawbacks of Anonymous Access-To-Object Types

Anonymous access-to-object types have important drawbacks. For example, some features that are available for named access types aren't available for the anonymous access types. Also, most of the drawbacks are related to how anonymous access-to-object types can potentially make the allocation and deallocation quite complicated or even error-prone.

For starters, some pool-related features aren't available for anonymous access-to-object types. For example, we cannot specify which pool is going to be used in the allocation of an anonymous access-to-object. In fact, the memory pool selection is compiler-dependent, so we cannot rely on an object being allocated from a specific pool when using **new** with an anonymous access-to-object type. (In contrast, as we know, each named access type has an associated pool, so objects allocated via **new** will be allocated from that pool.) Also, we cannot identify which pool was selected for the allocation of a specific object, so we don't have any information to use for the deallocation of that object.

Because the pool selection is hidden from us, this makes the memory deallocation more complicated. For example, we cannot instantiate the `Ada.Unchecked_Deallocation` procedure for anonymous access types. Also, some of the methods we could use to circumvent this limitation are error-prone, as we discuss in this section.

Also, storage-related features aren't available: specifying the storage size — especially, specifying that the access type has a storage size of zero — isn't possible.

Missing features

Let's see a code example that shows some of the features that aren't available for anonymous access-to-object types:

Listing 1471: missing_features.ads

```

1  with Ada.Unchecked_Deallocation;
2
3  package Missing_Features is
4
5      -- We cannot specify which pool will be used
6      -- in the anonymous access-to-object
7      -- allocation; the pool is selected by the
8      -- compiler:
9      IA : access Integer := new Integer;
10
11
12      -- All the features below aren't available
13      -- for an anonymous access-to-object:
14      --
15
16      -- Having a specific storage pool associated
17      -- with the access type:
18      type String_Access is
19          access String;
20      -- Automatically creates
21      -- String_Access'Storage_Pool
22
23      type Integer_Access is
24          access Integer
25          with Storage_Pool =>
26              String_Access'Storage_Pool;
27      -- ~~~~~
28      -- Using the pool from another
29      -- access type.
30
31      -- Specifying a deallocation function for the
32      -- access type:
33      procedure Free is
34          new Ada.Unchecked_Deallocation
35              (Object => Integer,
36               Name   => Integer_Access);
37
38      -- Specifying a limited storage size for
39      -- the access type:
40      type Integer_Access_Store_128 is
41          access Integer
42          with Storage_Size => 128;
43
44      -- Limiting the storage size for the
45      -- access type to zero:
46      type Integer_Access_Store_0 is
47          access Integer
48          with Storage_Size => 0;
49
50  end Missing_Features;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Missing_Anonymous_Access_To_Object_Features
 MD5: 87a5c1413a720da84fab414cf63236ec

In the `Missing_Features` package, we see some of the features that we cannot use for the anonymous `access Integer` type, but that are available for equivalent named access types:

- There's no specific memory pool associated with the access object IA. In contrast, named types — such as `String_Access` and `Integer_Access` — have an associated pool, and we can use the `Storage_Pool` aspect and the `Storage_Pool` attribute to customize them.
- We cannot instantiate the `Ada.Unchecked_Deallocation` procedure for the `access Integer` type. However, we can instantiate it for named access types such as the `Integer_Access` type.
- We cannot use the `Storage_Size` attribute for the `access Integer` type, but we're allowed to use it with named access types, which we do in the declaration of the `Integer_Access_Store_128` and `Integer_Access_Store_0` types.

Dangerous memory deallocation

We might think that we could make up for the absence of the `Ada.Unchecked_Deallocation` procedure for anonymous access-to-object types by converting those access objects (of anonymous access types) to a named type that has the same designated subtype. For example, if we have an access object IA of an anonymous `access Integer` type, we can convert it to the named `Integer_Access` type, provided this named access type is compatible with the anonymous access type, e.g.:

```
type Integer_Access is access all Integer
```

Let's see a complete code example:

Listing 1472: `show_dangerous_deallocation.adb`

```
1 with Ada.Unchecked_Deallocation;
2
3 procedure Show_Dangerous_Deallocation is
4   type Integer_Access is
5     access all Integer;
6
7   procedure Free is
8     new Ada.Unchecked_Deallocation
9       (Object => Integer,
10        Name   => Integer_Access);
11
12   IA : access Integer;
13 begin
14   IA := new Integer;
15   IA.all := 30;
16
17   -- Potentially erroneous deallocation via type
18   -- conversion:
19   Free (Integer_Access (IA));
20
21 end Show_Dangerous_Deallocation;
```

Code block metadata

Project: `Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_Erroneous`
MD5: `91e024a4338e2e4f8d5b308d95499c1c`

This example declares the IA access object of the anonymous `access Integer` type. After allocating an object for IA via `new`, we try to deallocate it by first converting it to the Inte-

ger_Access type, so that we can call the Free procedure to actually deallocate the object. Although this code compiles, it'll only work if both **access Integer** and Integer_Access types are using the same memory pool. Since we cannot really determine this, the result is potentially erroneous: it'll work if the compiler selected the same pool, but it'll fail otherwise.

Important

Because allocating memory for anonymous access types is potentially dangerous, we can use the No_Anonymous_Allocators restriction — which is available since Ada 2012 — to prevent this kind of memory allocation being used in the code. For example:

Listing 1473: show_dangerous_allocation.adb

```

1 pragma Restrictions (No_Anonymous_Allocators);
2
3 procedure Show_Dangerous_Allocation is
4   IA : access Integer;
5 begin
6   IA := new Integer;
7   IA.all := 30;
8 end Show_Dangerous_Allocation;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪Anonymous_Access_To_Object_Types.No_Anonymous_Allocators
 MD5: 0976821ce632f9635e33fd4f79c81ecd

Build output

```

show_dangerous_allocation.adb:6:10: error: violation of restriction "No_
  ↪Anonymous_Allocators" at line 1
gprbuild: *** compilation phase failed
```

Possible solution using named access types

A better solution to avoid issues when allocating and deallocating memory for anonymous access-to-object types is to allocate the object using a known pool. As mentioned before, the memory pool associated with a named access type is well-defined, so we can use this kind of types for memory allocation. In fact, we can use a named memory type to allocate an object via **new**, and then associate this allocated object with the access object of anonymous access type.

Let's see a code example:

Listing 1474: show_successful_deallocation.adb

```

1 with Ada.Unchecked_Deallocation;
2
3 procedure Show_Successful_Deallocation is
4
5   type Integer_Access is
6     access Integer;
7
8   procedure Free is
9     new Ada.Unchecked_Deallocation
10      (Object => Integer,
11       Name   => Integer_Access);
12
13   IA      : access Integer;
```

(continues on next page)

(continued from previous page)

```

14   Typed_IA : Integer_Access;
15
16   begin
17     Typed_IA := new Integer;
18     IA := Typed_IA;
19     IA.all := 30;
20
21     -- Deallocation of the access object that has
22     -- an associated type:
23     Free (Typed_IA);
24
25   end Show_Successful_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_1
MD5: eff8b54adfcc8cce10920dc3620ff1b9

In this example, all operations related to memory allocation are exclusively making use of the `Integer_Access` type, which is a named access type. In fact, `new Integer` allocates the object from the pool associated with the `Integer_Access` type, and the call to `Free` deallocates this object back into that pool. Therefore, associating this object with the `IA` access object — in the `IA := Typed_IA` assignment — doesn't create problems afterwards in the object's deallocation. (When calling `Free`, we only refer to the object of named access type, so the object is deallocated from a known pool.)

Of course, a potential issue here is that `IA` becomes a *dangling reference* (page 939) after the call to `Free`. Therefore, we can improve this solution by completely hiding the memory allocation and deallocation for the anonymous access types in subprograms — e.g. as part of a package. By doing so, we don't expose the named access type, thereby reducing the possibility of dangling references.

In fact, we can generalize this approach with the following (generic) package:

Listing 1475: `hidden_anonymous_allocation.ads`

```

1   generic
2     type T is private;
3   package Hidden_Anonymous_Allocation is
4
5     function New_T
6       return not null access T;
7
8     procedure Free (Obj : access T);
9
10  end Hidden_Anonymous_Allocation;

```

Listing 1476: `hidden_anonymous_allocation.adb`

```

1   with Ada.Unchecked_Deallocation;
2
3   package body Hidden_Anonymous_Allocation is
4
5     type T_Access is access all T;
6
7     procedure T_Access_Free is
8       new Ada.Unchecked_Deallocation
9         (Object => T,
10          Name   => T_Access);

```

(continues on next page)

(continued from previous page)

```

11
12  function New_T
13  return not null access T is
14  begin
15      return T_Access'(new T);
16      -- Using allocation of the T_Access type:
17      -- object is allocated from T_Access's pool
18  end New_T;
19
20  procedure Free (Obj : access T) is
21      Tmp : T_Access := T_Access (Obj);
22  begin
23      T_Access_Free (Tmp);
24      -- Using deallocation procedure of the
25      -- T_Access type
26  end Free;
27
28  end Hidden_Anonymous_Allocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Hidden_Alloc_Dealloc_Anonymous_Access_To_Object
MD5: bd3831829f34f06a1d3c25a975c850a3

In the generic `Hidden_Anonymous_Allocation` package, `New_T` allocates a new object internally and returns an anonymous access to this object. The `Free` procedure deallocates this object.

In the body of the `Hidden_Anonymous_Allocation` package, we use the named access type `T_Access` to handle the actual memory allocation and deallocation. As expected, because those operations happen on the pool associated with the `T_Access` type, we don't have to worry about potential deallocation issues.

Finally, we can instantiate this package for the type we want to have anonymous access types for, say a type named `Rec`. Then, when using the `Rec` type in the main subprogram, we can simply call the corresponding subprograms for memory allocation and deallocation. For example:

Listing 1477: info.ads

```

1  with Hidden_Anonymous_Allocation;
2
3  package Info is
4
5      type Rec is private;
6
7      function New_Rec return not null access Rec;
8
9      procedure Free (Obj : access Rec);
10
11  private
12
13      type Rec is record
14          I : Integer;
15      end record;
16
17      package Rec_Allocation is new
18          Hidden_Anonymous_Allocation (T => Rec);
19
20      function New_Rec return not null access Rec

```

(continues on next page)

(continued from previous page)

```

21     renames Rec_Allocation.New_T;
22
23     procedure Free (Obj : access Rec)
24         renames Rec_Allocation.Free;
25
26 end Info;

```

Listing 1478: show_info_allocation_deallocation.adb

```

1  with Info; use Info;
2
3  procedure Show_Info_Allocation_Deallocation is
4      RA : constant not null access Rec := New_Rec;
5  begin
6      Free (RA);
7  end Show_Info_Allocation_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Hidden_Alloc_Dealloc_Anonymous_Access_To_Object
 MD5: d71e8ed70e280c6d5d9fc2d49c1eb6c3

In this example, we instantiate the Hidden_Anonymous_Allocation package in the Info package, which also defines the Rec type. We associate the New_T and Free subprograms with the Rec type by using subprogram renaming. Finally, in the Show_Info_Allocation_Deallocation procedure, we use these subprograms to allocate and deallocate the type.

Possible solution using the stack

Another approach that we could consider to avoid memory deallocation issues for anonymous access-to-object types is by simply using the stack for the object creation. For example:

Listing 1479: show_automatic_deallocation.adb

```

1  procedure Show_Automatic_Deallocation is
2      I : aliased Integer;
3      -- ^ Allocating object on the stack
4
5      IA : access Integer;
6  begin
7      IA := I'Access;
8      -- Indirect allocation:
9      -- object creation on the stack.
10
11     IA.all := 30;
12
13     -- Automatic deallocation at the end of the
14     -- procedure because the integer variable is
15     -- on the stack.
16 end Show_Automatic_Deallocation;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Object_Types.Deallocation_Anonymous_Access_To_Object_2
 MD5: 4381db8ba87717978a9629b1e6a5f1fc

In this case, we create the I object on the stack by simply declaring it. Then, we get access to it and assign it to the IA access object.

With this approach, we're indirectly allocating an object for an anonymous access type by creating it on the stack. Also, because we know that the I is automatically deallocated when it gets out of scope, we don't have to worry about explicitly deallocating the object referred by IA.

When to use anonymous access-to-objects types

In summary, anonymous access-to-object types have many drawbacks that often outweigh *their benefits* (page 998). In fact, allocation for those types can quickly become very complicated. Therefore, in general, they're not a good alternative to named access types. Indeed, the difficulties that we've just seen might make them a much worse option than just using named access types instead.

We might consider using anonymous access-to-objects types only in cases when we reach a point in our implementation work where using named access types becomes impossible — or when using them becomes even more complicated than equivalent solutions using anonymous access types. This scenario, however, is usually the exception rather than the rule. Thus, as a general guideline, we should always aim to use named access types.

That being said, an important exception to this advice is when we're *interfacing to other languages* (page 1024). In this case, as we'll discuss later, using anonymous access-to-objects types can be significantly simpler (compared to named access types) without the drawbacks that we've just discussed.

28.2.3 Access discriminants

Previously, we've discussed *discriminants as access values* (page 891). In that section, we only used named access types. Now, in this section, we see how to use anonymous access types as discriminants. This feature is also known as *access discriminants* and it provides some flexibility that can be interesting in terms of software design, as we'll discuss later.

Let's start with an example:

Listing 1480: custom_recs.ads

```

1 package Custom_Recs is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type:
5   type Rec (IA : access Integer) is record
6     I : Integer := IA.all;
7   end record;
8
9   procedure Show (R : Rec);
10
11 end Custom_Recs;
```

Listing 1481: custom_recs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Recs is
4
5   procedure Show (R : Rec) is
6   begin
7     Put_Line ("R.IA = "
8               & Integer'Image (R.IA.all));
9     Put_Line ("R.I = "
```

(continues on next page)

(continued from previous page)

```

10         & Integer'Image (R.I));
11     end Show;
12
13 end Custom_Recs;

```

Listing 1482: show_access_discriminants.adb

```

1  with Custom_Recs; use Custom_Recs;
2
3  procedure Show_Access_Discriminants is
4      I : aliased Integer := 10;
5      R : Rec (I'Access);
6  begin
7      Show (R);
8
9      I := 20;
10     R.I := 30;
11     Show (R);
12 end Show_Access_Discriminants;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_Discriminants.Simple_Example
MD5: f8e127fda4f7ea0f1593165d6a966df6

Runtime output

```

R.IA = 10
R.I  = 10
R.IA = 20
R.I  = 30

```

In this example, we use an anonymous access type for the discriminant in the declaration of the Rec type of the Custom_Recs package. In the Show_Access_Discriminants procedure, we declare R and provide access to the local I integer.

Similarly, we can use unconstrained designated subtypes:

Listing 1483: persons.ads

```

1  package Persons is
2
3      -- Declaring a discriminant with an anonymous
4      -- access type whose designated subtype is
5      -- unconstrained:
6      type Person (Name : access String) is record
7          Age : Integer;
8      end record;
9
10     procedure Show (P : Person);
11
12 end Persons;

```

Listing 1484: persons.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Persons is
4

```

(continues on next page)

(continued from previous page)

```

5  procedure Show (P : Person) is
6  begin
7      Put_Line ("Name = "
8              & P.Name.all);
9      Put_Line ("Age = "
10             & Integer'Image (P.Age));
11  end Show;
12
13 end Persons;

```

Listing 1485: show_person.adb

```

1  with Persons; use Persons;
2
3  procedure Show_Person is
4      S : aliased String := "John";
5      P : Person (S'Access);
6  begin
7      P.Age := 30;
8      Show (P);
9  end Show_Person;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_Discriminants.Persons
 MD5: f0149d572e0ec192476836bfd00dd9e

Runtime output

```

Name = John
Age  = 30

```

In this example, for the discriminant of the Person type, we use an anonymous access type whose designated subtype is unconstrained. In the Show_Person procedure, we declare the P object and provide access to the S string.

** In the Ada Reference Manual**

- [3.7 Discriminants](#)²⁷⁹
- [3.10.2 Operations of Access Types](#)²⁸⁰

Default Value of Access Discriminants

In contrast to named access types, we cannot use a default value for the access discriminant of a non-limited type:

Listing 1486: custom_recs.ads

```

1  package Custom_Recs is
2
3      -- Declaring a discriminant with an anonymous
4      -- access type and a default value:
5      type Rec (IA : access Integer :=
6              new Integer'(0)) is

```

(continues on next page)

²⁷⁹ <http://www.ada-auth.org/standards/22rm/html/RM-3-7.html>

²⁸⁰ <http://www.ada-auth.org/standards/22rm/html/RM-3-10-2.html>

(continued from previous page)

```
7   record
8     I : Integer := IA.all;
9   end record;
10
11 end Custom_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_Discriminants.Default_Expression_Non_Limited_Type
MD5: c3ddf1cdfdaefa873ad66b9e47e03058

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its
↳ associated owner is deallocated [enabled by default]
custom_recs.ads:6:21: error: (Ada 2005) access discriminants of nonlimited types
↳ cannot have defaults
gprbuild: *** compilation phase failed
```

However, if we change the type declaration to be a limited type, having a default value for the access discriminant is OK:

Listing 1487: custom_recs.ads

```
1 package Custom_Recs is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type and a default value:
5   type Rec (IA : access Integer :=
6             new Integer'(0)) is limited
7
8   record
9     I : Integer := IA.all;
10  end record;
11
12  procedure Show (R : Rec);
13
14 end Custom_Recs;
```

Listing 1488: custom_recs.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Custom_Recs is
4
5   procedure Show (R : Rec) is
6   begin
7     Put_Line ("R.IA = "
8              & Integer'Image (R.IA.all));
9     Put_Line ("R.I = "
10             & Integer'Image (R.I));
11  end Show;
12
13 end Custom_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_Discriminants.Default_Expression_Limited_Type
MD5: ae872f1dec64b8e955f04789ca4db218

Build output

```
custom_recs.ads:6:21: warning: coextension will not be deallocated when its
↳associated owner is deallocated [enabled by default]
```

Note that, if we don't provide a value for the access discriminant when declaring an object R, the default value is allocated (via **new**) during R's creation.

Listing 1489: show_access_discriminants.adb

```
1 with Custom_Recs; use Custom_Recs;
2
3 procedure Show_Access_Discriminants is
4   R : Rec;
5   -- ^^
6   -- This triggers "new Integer'(0)", so an
7   -- integer object is allocated and stored in
8   -- the R.IA discriminant.
9 begin
10  Show (R);
11
12  -- R gets out of scope here, and the object
13  -- allocated via new hasn't been deallocated.
14 end Show_Access_Discriminants;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
↳Discriminants.Default_Expression_Limited_Type
MD5: f5d9dee26044ccab2193ab419638de79
```

Build output

```
show_access_discriminants.adb:4:04: warning: coextension will not be deallocated
↳when its associated owner is deallocated [enabled by default]
custom_recs.ads:6:21: warning: coextension will not be deallocated when its
↳associated owner is deallocated [enabled by default]
```

Runtime output

```
R.IA = 0
R.I  = 0
```

In this case, the allocated object won't be deallocated when R gets out of scope!

Benefits of Access Discriminants

Access discriminants have the same benefits that we've already seen earlier while discussing *discriminants as access values* (page 891). An additional benefit is its extended flexibility: access discriminants are compatible with any access T'Access, as long as T is of the designated subtype.

Consider the following example using the named access type Access_String:

Listing 1490: persons.ads

```
1 package Persons is
2
3   type Access_String is access all String;
4
5   -- Declaring a discriminant with a named
6   -- access type:
```

(continues on next page)

(continued from previous page)

```
7  type Person (Name : Access_String) is record
8      Age : Integer;
9  end record;
10
11  procedure Show (P : Person);
12
13  end Persons;
```

Listing 1491: persons.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Persons is
4
5      procedure Show (P : Person) is
6      begin
7          Put_Line ("Name = "
8                  & P.Name.all);
9          Put_Line ("Age = "
10                 & Integer'Image (P.Age));
11      end Show;
12
13  end Persons;
```

Listing 1492: show_person.adb

```
1  with Persons; use Persons;
2
3  procedure Show_Person is
4      S : aliased String := "John";
5      P : Person (S'Access);
6      --      ^^^^^^ ERROR: cannot use local
7      --              object
8      --
9      --  We can, however, allocate the string via
10     --  new:
11     --
12     --  S : Access_String := new String'("John");
13     --  P : Person (S);
14  begin
15      P.Age := 30;
16      Show (P);
17  end Show_Person;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_
Discriminants.Persons
MD5: e918db3790c7ffeeb7c0f54ced9f48b9

Build output

```
show_person.adb:5:16: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed
```

This code doesn't compile because we cannot have a non-local pointer (`Access_String`) pointing to the local object `S`. The only way to make this work is by allocating the string via `new` (i.e.: `S : Access_String := new String`).

However, if we use an access discriminant in the declaration of `Person`, the code compiles fine:

Listing 1493: persons.ads

```

1 package Persons is
2
3   -- Declaring a discriminant with an anonymous
4   -- access type:
5   type Person (Name : access String) is record
6     Age : Integer;
7   end record;
8
9   procedure Show (P : Person);
10
11 end Persons;
```

Listing 1494: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4   S : aliased String := "John";
5   P : Person (S'Access);
6   --      ^^^^^^^ OK
7
8   -- Allocating the string via new and using it
9   -- in P's declaration is OK as well, but we
10  -- should manually deallocate it before S
11  -- gets out of scope:
12  --
13  -- S : access String := new String("John");
14  -- P : Person (S);
15 begin
16   P.Age := 30;
17   Show (P);
18 end Show_Person;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_

 ↪Discriminants.Persons
 MD5: 6516fb4e0cbbac9cfe07a56e48ea9ff3

Runtime output

```

Name = John
Age  = 30
```

In this case, getting access to the local object S and using it for P's discriminant is perfectly fine.

Preventing dangling pointers

Note that the usual rules that prevent dangling pointers still apply here. This ensures that we can safely use access discriminants. For example:

Listing 1495: show_person.adb

```

1 with Persons; use Persons;
2
3 procedure Show_Person is
4
```

(continues on next page)

(continued from previous page)

```

5  function Local_Init return Person is
6      S : aliased String := "John";
7  begin
8      return (Name => S'Access, Age => 30);
9      --      ~~~~~
10     --      ERROR: dangling reference!
11 end Local_Init;
12
13 P : Person := Local_Init;
14 begin
15     Show (P);
16 end Show_Person;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Access_Discriminants.Persons
MD5: 9c8d2aebf60b8bb19e455cb6bc5730eb

Build output

```

show_person.adb:8:07: error: access discriminant in return object would be a
↳dangling reference
gprbuild: *** compilation phase failed

```

In this example, compilation fails in the `Local_Init` function when trying to return an object of `Person` type because `S'Access` would be a dangling reference.

28.2.4 Self-reference

Previously, we've seen that we can declare *self-references* (page 908) using named access types. We can do the same with anonymous access types. Let's revisit the code example that implements linked lists:

Listing 1496: linked_lists.ads

```

1  generic
2      type T is private;
3  package Linked_Lists is
4
5      type List is limited private;
6
7      procedure Append_Front
8          (L : in out List;
9           E : T);
10
11     procedure Append_Rear
12         (L : in out List;
13          E : T);
14
15     procedure Show (L : List);
16
17 private
18
19     type Component is record
20         Next : access Component;
21         --      ~~~~~
22         --      Self-reference
23         --
24         --      (Note that we haven't finished the

```

(continues on next page)

(continued from previous page)

```

25      --      declaration of the "Component" type
26      --      yet, but we're already referring to
27      --      it.)
28
29      Value : T;
30  end record;
31
32  type List is access all Component;
33
34  end Linked_Lists;

```

Listing 1497: linked_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Linked_Lists is
4
5      procedure Append_Front
6          (L : in out List;
7           E :          T)
8      is
9          New_First : constant List := new
10             Component'(Value => E,
11                        Next => L);
12      begin
13          L := New_First;
14      end Append_Front;
15
16      procedure Append_Rear
17          (L : in out List;
18           E :          T)
19      is
20          New_Last : constant List := new
21             Component'(Value => E,
22                        Next => null);
23      begin
24          if L = null then
25              L := New_Last;
26          else
27              declare
28                  Last : List := L;
29              begin
30                  while Last.Next /= null loop
31                      Last := List (Last.Next);
32                      --      ^^^
33                      --      type conversion:
34                      --      "access Component" to
35                      --      "List"
36                  end loop;
37                  Last.Next := New_Last;
38              end;
39          end if;
40      end Append_Rear;
41
42      procedure Show (L : List) is
43          Curr : List := L;
44      begin
45          if L = null then
46              Put_Line ("[ ]");
47          else
48              Put ("[" );

```

(continues on next page)

(continued from previous page)

```

49     loop
50         Put (Curr.Value'Image);
51         Put (" ");
52         exit when Curr.Next = null;
53         Curr := Curr.Next;
54     end loop;
55     Put_Line ("]");
56 end if;
57 end Show;
58
59 end Linked_Lists;

```

Listing 1498: test_linked_list.adb

```

1  with Linked_Lists;
2
3  procedure Test_Linked_List is
4      package Integer_Lists is new
5          Linked_Lists (T => Integer);
6      use Integer_Lists;
7
8      L : List;
9  begin
10     Append_Front (L, 3);
11     Append_Rear (L, 4);
12     Append_Rear (L, 5);
13     Append_Front (L, 2);
14     Append_Front (L, 1);
15     Append_Rear (L, 6);
16     Append_Rear (L, 7);
17
18     Show (L);
19 end Test_Linked_List;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Self_
 ↪Reference.Linked_List_Example
 MD5: 98b9b2ce6fac3064326e6345520dc650

Runtime output

```
[ 1  2  3  4  5  6  7 ]
```

Here, in the declaration of the Component type (in the private part of the generic `Linked_Lists` package), we declare `Next` as an anonymous access type that refers to the Component type. (Note that at this point, we haven't finished the declaration of the Component type yet, but we're already using it as the designated subtype of an anonymous access type.) Then, we declare `List` as a general access type (with `Component` as the designated subtype).

It's worth mentioning that the `List` type and the anonymous **access** Component type aren't the same type, although they share the same designated subtype. Therefore, in the implementation of the `Append_Rear` procedure, we have to use type conversion to convert from the anonymous **access** Component type to the (named) `List` type.

28.2.5 Mutually dependent types using anonymous access types

In the section on *mutually dependent types using access types* (page 911), we've seen a code example that was using named access types. We could now rewrite it using anonymous access types:

Listing 1499: mutually_dependent.ads

```

1 package Mutually_Dependent is
2
3     type T2;
4
5     type T1 is record
6         B : access T2;
7     end record;
8
9     type T2 is record
10        A : access T1;
11    end record;
12
13 end Mutually_Dependent;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Mutually_
 ↳Dependent_Anonymous_Access_Types.Example
 MD5: 09f869d99b9c16882554588bb806a113

In this example, T1 and T2 are mutually dependent types. We're using anonymous access types in the declaration of the B and A components.

28.2.6 Access parameters

In the previous chapter, we talked about *parameters as access values* (page 898). As you might have expected, we can also use anonymous access types as parameters of a subprogram. However, they're limited to be **in** parameters of a subprogram or return type of a function (also called the access result type):

Listing 1500: names.ads

```

1 package Names is
2
3     function Init (S1, S2 : String)
4         return access String;
5         ~~~~~
6     -- Anonymous access type as the access
7     -- result type.
8
9     procedure Show (N : access constant String);
10    -- ~~~~~
11    -- Anonymous access type as a parameter type.
12
13 end Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
 ↳Access_Parameters.Names
 MD5: 622a76c4b133ed2715f18c175694cbe2

In this example, we have a string as the access result type of the Init function, and another string as the access parameter of the Show procedure.

This is the complete code example:

Listing 1501: names.ads

```
1 package Names is
2
3     function Init (S1, S2 : String)
4         return access String;
5
6     procedure Show (N : access constant String);
7
8 private
9
10    function Init (S1, S2 : String)
11        return access String is
12        (new String'(S1 & "-" & S2));
13
14 end Names;
```

Listing 1502: names.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5     procedure Show (N : access constant String) is
6     begin
7         Put_Line ("Name: " & N.all);
8     end Show;
9
10 end Names;
```

Listing 1503: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4     N : access String := Init ("Lily", "Ann");
5 begin
6     Show (N);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
Access_Parameters.Names
MD5: 9fe629f29de2898f2b82d9146b22fd1a

Runtime output

Name: Lily-Ann

Note that we're not using the **in** parameter mode in the Show procedure above. Usually, this parameter mode can be omitted, as it is the default parameter mode — **procedure P (I : Integer)** is the same as **procedure P (I : in Integer)**. However, in the case of the Show procedure, the **in** parameter mode isn't just optionally absent. In fact, for access parameters, the parameter mode is always implied as **in**, so writing it explicitly is actually forbidden. In other words, we can only write **N : access String** or **N : access constant String**, but we cannot write **N : in access String** or **N : in access constant String**.

For further reading...

When we discussed *parameters as access values* (page 898) in the previous chapter, we saw how we can simply use different parameter modes to write a program instead of using access types. Basically, to implement the same functionality, we just replaced the access types by selecting the correct parameter modes instead and used *simpler* data types.

Let's do the same exercise again, this time by adapting the previous code example with anonymous access types:

Listing 1504: names.ads

```

1 package Names is
2
3     function Init (S1, S2 : String)
4         return String;
5
6     procedure Show (N : String);
7
8 private
9
10    function Init (S1, S2 : String)
11        return String is
12        (S1 & "-" & S2);
13
14 end Names;
```

Listing 1505: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Names is
4
5     procedure Show (N : String) is
6     begin
7         Put_Line ("Name: " & N);
8     end Show;
9
10 end Names;
```

Listing 1506: show_names.adb

```

1 with Names; use Names;
2
3 procedure Show_Names is
4     N : String := Init ("Lily", "Ann");
5 begin
6     Show (N);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 Anonymous_Access_Parameters.Names_String
 MD5: 643f193999ef8de9bcefb11d9bdd21d7

Runtime output

Name: Lily-Ann

Although we're using simple strings instead of access types in this version of the code example, we're still getting a similar behavior. However, there is a small, yet important difference in the way the string returned by Init is being allocated: while the previous

implementation (which was using an access result type) was allocating the string on the heap, we're now allocating the string on the stack.

Later on, we talk about the *accessibility rules in the case of access parameters* (page 1043).

In general, we should avoid access parameters whenever possible and simply use objects and parameter modes directly, as it makes the design simpler and less error-prone. One exception is when we're interfacing to other languages, especially C: this is our *next topic* (page 1024). Another time when access parameters are vital is for inherited primitive operations for tagged types. We discuss this *later on* (page 1027).

In the Ada Reference Manual

- 3.10 Access Types²⁸¹

Interfacing To Other Languages

We can use access parameters to interface to other languages. This can be particularly useful when interfacing to C code that makes use of pointers. For example, let's assume we want to call the `add_one` function below in our Ada implementation:

Listing 1507: operations_c.h

```
1 void add_one(int *p_i);
```

Listing 1508: operations_c.c

```
1 void add_one(int *p_i)
2 {
3     *p_i = *p_i + 1;
4 }
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_Parameters.C_Interfacing
MD5: 3270f3b2415266a203a6f4c605c3831b

We could map the `int *` parameter of `add_one` to `access Integer` in the Ada specification:

```
procedure Add_One (IA : access Integer)
with Import, Convention => C;
```

This is a complete code example:

Listing 1509: operations.ads

```
1 package Operations is
2
3     procedure Add_One (IA : access Integer)
4         with Import, Convention => C;
5
6 end Operations;
```

²⁸¹ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Listing 1510: show_operations.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Operations; use Operations;
4
5 procedure Show_Operations is
6   I : aliased Integer := 42;
7 begin
8   Put_Line (I'Image);
9   Add_One (I'Access);
10  Put_Line (I'Image);
11 end Show_Operations;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
 ↪Access_Parameters.C_Interfacing
 MD5: 0219acdbd2dad69962875199ffdd930e

Once again, we can replace access parameters with simpler types by using the appropriate parameter mode. In this case, we could replace **access Integer** by **aliased in out Integer**. This is the modified version of the code:

Listing 1511: operations.ads

```

1 package Operations is
2
3   procedure Add_One
4     (IA : aliased in out Integer)
5     with Import, Convention => C;
6
7 end Operations;

```

Listing 1512: show_operations.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Operations; use Operations;
4
5 procedure Show_Operations is
6   I : aliased Integer := 42;
7 begin
8   Put_Line (I'Image);
9   Add_One (I);
10  Put_Line (I'Image);
11 end Show_Operations;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
 ↪Access_Parameters.C_Interfacing
 MD5: 2c5a81b8d77f0fff8a73f7912be6b6fe

However, there are situations where aliased objects cannot be used. For example, suppose we want to allocate memory inside a C function. In this case, the pointer to that memory block must be mapped to an access type in Ada.

Let's extend the previous C code example and introduce the `alloc_integer` and `dealloc_integer` functions, which allocate and deallocate an integer value:

Listing 1513: operations_c.h

```
1 int * alloc_integer();
2
3 void dealloc_integer(int *p_i);
4
5 void add_one(int *p_i);
```

Listing 1514: operations_c.c

```
1 #include <stdlib.h>
2
3 int * alloc_integer()
4 {
5     return malloc(sizeof(int));
6 }
7
8 void dealloc_integer(int *p_i)
9 {
10    free (p_i);
11 }
12
13 void add_one(int *p_i)
14 {
15     *p_i = *p_i + 1;
16 }
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: ec6dea12d0a948489cce21b0cc0a1ad2

In this case, we really have to use access types to interface to these C functions. In fact, we need an access result type to interface to the `alloc_integer()` function, and an access parameter in the case of the `dealloc_integer()` function. This is the corresponding specification in Ada:

Listing 1515: operations.ads

```
1 package Operations is
2
3     function Alloc_Integer return access Integer
4         with Import, Convention => C;
5
6     procedure Dealloc_Integer (IA : access Integer)
7         with Import, Convention => C;
8
9     procedure Add_One
10         (IA : aliased in out Integer)
11         with Import, Convention => C;
12
13 end Operations;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↪Access_Parameters.C_Interfacing
MD5: bcbc8a87037b64fc6469e67b928e6172

Note that we're still using an aliased integer type for the `Add_One` procedure, while we're using access types for the other two subprograms.

Finally, as expected, we can use this specification in a test application:

Listing 1516: show_operations.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Operations; use Operations;
4
5 procedure Show_Operations is
6   I : access Integer := Alloc_Integer;
7 begin
8   I.all := 42;
9   Put_Line (I.all'Image);
10
11   Add_One (I.all);
12   Put_Line (I.all'Image);
13
14   Dealloc_Integer (I);
15 end Show_Operations;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
 ↪Access_Parameters.C_Interfacing
 MD5: b2b96a166926528bc44059b56e31fb55

In this application, we get a C pointer from the `alloc_integer` function and encapsulate it in an Ada access type, which we then assign to `I`. In the last line of the procedure, we call `Dealloc_Integer` and pass `I` to it, which deallocates the memory block indicated by the C pointer.

In the Ada Reference Manual

- [3.10 Access Types](#)²⁸²

Inherited Primitive Operations For Tagged Types

In order to declare inherited primitive operations for tagged types that use access types, we need to use access parameters. The reason is that, to be a primitive operation for some tagged type — and hence inheritable — the subprogram must reference the tagged type name directly in the parameter profile. This means that a named access type won't suffice, because only the access type name would appear in the profile. For example:

Listing 1517: inherited_primitives.ads

```

1 package Inherited_Primitives is
2
3   type T is tagged private;
4
5   type T_Access is access all T;
6
7   procedure Proc (N : T_Access);
8   -- Proc is not a primitive of type T.
9
10  type T_Child is new T with private;
11
12  type T_Child_Access is access all T_Child;
```

(continues on next page)

²⁸² <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

14 private
15
16     type T is tagged null record;
17
18     type T_Child is new T with null record;
19
20 end Inherited_Primitives;

```

Listing 1518: inherited_primitives.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inherited_Primitives is
4
5     procedure Proc (N : T_Access) is null;
6
7 end Inherited_Primitives;

```

Listing 1519: show_inherited_primitives.adb

```

1 with Inherited_Primitives;
2 use Inherited_Primitives;
3
4 procedure Show_Inherited_Primitives is
5     Obj      : T_Access      := new T;
6     Obj_Child : T_Child_Access := new T_Child;
7 begin
8     Proc (Obj);
9     Proc (Obj_Child);
10    -- ^^^^^^^^^^
11    -- ERROR: Proc is not inherited!
12 end Show_Inherited_Primitives;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
↳ Access_Parameters.Inherited_Primitives
MD5: 8235b21caa9f1f105f533d74d891adfe

Build output

```

show_inherited_primitives.adb:9:10: error: expected type "T_Access" defined at↳
↳ inherited_primitives.ads:5
show_inherited_primitives.adb:9:10: error: found type "T_Child_Access" defined at↳
↳ inherited_primitives.ads:12
gprbuild: *** compilation phase failed

```

In this example, Proc is not a primitive of type T because it's referring to type T_Access, not type T. This means that Proc isn't inherited when we derive the T_Child type. Therefore, when we call Proc (Obj_Child), a compilation error occurs because the compiler expects type T_Access — there's no Proc (N : T_Child_Access) that could be used here.

If we replace T_Access in the Proc procedure with an an access parameter (**access** T), the subprogram becomes a primitive of T:

Listing 1520: inherited_primitives.ads

```

1 package Inherited_Primitives is
2
3     type T is tagged private;

```

(continues on next page)

(continued from previous page)

```

4  procedure Proc (N : access T);
5  -- Proc is a primitive of type T.
6
7  type T_Child is new T with private;
8
9  private
10
11     type T is tagged null record;
12
13     type T_Child is new T with null record;
14
15 end Inherited_Primitives;
16

```

Listing 1521: inherited_primitives.adb

```

1  package body Inherited_Primitives is
2
3     procedure Proc (N : access T) is null;
4
5 end Inherited_Primitives;

```

Listing 1522: show_inherited_primitives.adb

```

1  with Inherited_Primitives;
2  use Inherited_Primitives;
3
4  procedure Show_Inherited_Primitives is
5     Obj      : access T      := new T;
6     Obj_Child : access T_Child := new T_Child;
7  begin
8     Proc (Obj);
9     Proc (Obj_Child);
10    --      ^^^^^^^^^
11    --      OK: Proc is inherited!
12 end Show_Inherited_Primitives;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_
 ↪Access_Parameters.Inherited_Primitives
 MD5: a7e9b8bc92e346758cc4ade43bb4b02d

Now, the child type T_Child (derived from the T) inherits the primitive operation Proc. This inherited operation has an access parameter designating the child type:

```

type T_Child is new T with private;

procedure Proc (N : access T_Child);
-- Implicitly inherited primitive operation

```

** In the Ada Reference Manual**

- 3.9.2 Dispatching Operations of Tagged Types²⁸³

²⁸³ <http://www.ada-auth.org/standards/22rm/html/RM-3-9-2.html>

28.2.7 User-Defined References

Implicit dereferencing (page 913) isn't limited to the contexts that Ada supports by default: we can also add implicit dereferencing to our own types by using the `Implicit_Dereference` aspect.

To do this, we have to declare:

- a reference type, where we use the `Implicit_Dereference` aspect to specify the reference discriminant, which is the record discriminant that will be dereferenced; and
- a reference object, which contains an access value that will be dereferenced.

Also, for the reference type, we have to:

- specify the reference discriminant as an *access discriminant* (page 1011); and
- indicate the name of the reference discriminant when specifying the `Implicit_Dereference` aspect.

Let's see a simple example:

Listing 1523: `show_user_defined_reference.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_User_Defined_Reference is
4
5      type Id_Number is record
6          Id : Positive;
7      end record;
8
9      --
10     -- Reference type:
11     --
12     type Id_Ref (Ref : access Id_Number) is
13         -- ^ reference discriminant
14         null record
15             with Implicit_Dereference => Ref;
16         --      ^^^
17         --      name of the reference
18         --      discriminant
19
20     --
21     -- Access value:
22     --
23     I : constant access Id_Number :=
24         new Id_Number'(Id => 42);
25
26     --
27     -- Reference object:
28     --
29     R : Id_Ref (I);
30 begin
31     Put_Line ("ID: "
32              & Positive'Image (R.Id));
33     --      ^ Equivalent to:
34     --      R.Ref.Id
35     --      or:
36     --      R.Ref.all.Id
37 end Show_User_Defined_Reference;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.Simple_User_Defined_References
 MD5: 33eaa7e8e75b4eb56d64dcc17e2932aa

Runtime output

ID: 42

Here, we declare a simple record type (`Id_Number`) and a corresponding reference type (`Id_Ref`). Note that:

- the reference discriminant `Ref` has an access to the `Id_Number` type; and
- we indicate this reference discriminant in the `Implicit_Dereference` aspect.

Then, we declare an access value (the `I` constant) and use it for the `Ref` discriminant in the declaration of the reference object `R`.

Finally, we implicitly dereference `R` and access the `Id` component by simply writing `R.Id` — instead of the extended forms `R.Ref.Id` or `R.Ref.all.Id`.

Important

The extended form mentioned in the example that we just saw (`R.Ref.all.Id`) makes it clear that two steps happen when evaluating `R.Id`:

- First, `R.Ref` is implied from `R` because of the `Implicit_Dereference` aspect.
- Then, `R.Ref` is implicitly dereferenced to `R.Ref.all`.

After these two steps, we can access the actual object. (In our case, we can access the `Id` component.)

Note that we cannot use access types directly for the reference discriminant. For example, if we made the following change in the previous code example, it wouldn't compile:

```
type Id_Number_Access is access Id_Number;

-- Reference type:
type Id_Ref (Ref : Id_Number_Access) is
--      ^ ERROR: it must be
--      an access
--      discriminant!
null record
  with Implicit_Dereference => Ref;
```

However, we could use other forms — such as `not null access` — in the reference discriminant:

```
-- Reference type:
type Id_Ref (Ref : not null access Id_Number) is
null record
  with Implicit_Dereference => Ref;
```

In the Ada Reference Manual

- 4.1.5 User-Defined References²⁸⁴

²⁸⁴ <http://www.ada-auth.org/standards/22rm/html/RM-4-1-5.html>

Dereferencing of tagged types

Naturally, implicit dereferencing is also possible when calling primitives of a tagged type. For example, let's change the declaration of the `Id_Number` type from the previous code example and add a `Show` primitive.

Listing 1524: info.ads

```
1 package Info is
2   type Id_Number (Id : Positive) is
3     tagged private;
4
5   procedure Show (R : Id_Number);
6 private
7   type Id_Number (Id : Positive) is
8     tagged null record;
9 end Info;
```

Listing 1525: info.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Info is
4
5   procedure Show (R : Id_Number) is
6   begin
7     Put_Line ("ID: " & Positive'Image (R.Id));
8   end Show;
9
10 end Info;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.Dereferencing_Tagged_Types
MD5: 4de65094963450dc3a7505dbf93c2551

Then, let's declare a reference type and a reference object in the test application:

Listing 1526: show_user_defined_reference.adb

```
1 with Info; use Info;
2
3 procedure Show_User_Defined_Reference is
4
5   -- Reference type:
6   type Id_Ref (Ref : access Id_Number) is
7     null record
8     with Implicit_Dereference => Ref;
9
10  -- Access value:
11  I : constant access Id_Number :=
12    new Id_Number (42);
13
14  -- Reference object:
15  R : Id_Ref (I);
16 begin
17
18  R.Show;
19  -- Equivalent to:
20  -- R.Ref.all.Show;
21
```

(continues on next page)

(continued from previous page)

```
22 end Show_User_Defined_Reference;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.Dereferencing_Tagged_Types
 MD5: 9c5dfc4f2b8e085efde9e61689243f70

Runtime output

ID: 42

Here, we can call the Show procedure by simply writing R.Show instead of R.Ref.all.Show.

Simple container

A typical application of user-defined references is to create cursors when iterating over a container. As an example, let's implement the National_Date_Info package to store the national day of a country:

Listing 1527: national_date_info.ads

```
1 package National_Date_Info is
2
3   subtype Country_Code is String (1 .. 3);
4
5   type Time is record
6     Year  : Integer;
7     Month : Positive range 1 .. 12;
8     Day   : Positive range 1 .. 31;
9   end record;
10
11  type National_Date is tagged record
12    Country : Country_Code;
13    Date    : Time;
14  end record;
15
16  type National_Date_Access is
17    access National_Date;
18
19  procedure Show (Nat_Date : National_Date);
20
21 end National_Date_Info;
```

Listing 1528: national_date_info.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body National_Date_Info is
4
5   procedure Show (Nat_Date : National_Date) is
6   begin
7     Put_Line ("Country: "
8              & Nat_Date.Country);
9     Put_Line ("Year: "
10             & Integer'Image
11             (Nat_Date.Date.Year));
12   end Show;
13
14 end National_Date_Info;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.National_Dates
 MD5: 90fd6740d701025e1d5f30c9751a528d

Here, National_Date is a record type that we use to store the national day information. We can call the Show procedure to display this information.

Now, let's implement the National_Date_Containers with a container for national days:

Listing 1529: national_date_containers.ads

```

1 with National_Date_Info; use National_Date_Info;
2
3 package National_Date_Containers is
4
5     -- Reference type:
6     type National_Date_Reference
7       (Ref : access National_Date) is
8       tagged limited null record
9       with Implicit_Dereference => Ref;
10
11     -- Container (as an array):
12     type National_Dates is
13       array (Positive range <>) of
14         National_Date_Access;
15
16     -- The Find function scans the container to
17     -- find a specific country, which is returned
18     -- as a reference object.
19     function Find (Nat_Dates : National_Dates;
20                   Country   : Country_Code)
21                   return National_Date_Reference;
22
23 end National_Date_Containers;
```

Listing 1530: national_date_containers.adb

```

1 package body National_Date_Containers is
2
3     function Find (Nat_Dates : National_Dates;
4                   Country   : Country_Code)
5                   return National_Date_Reference
6
7     is
8     begin
9         for I in Nat_Dates'Range loop
10             if Nat_Dates (I).Country = Country then
11                 return National_Date_Reference'(
12                     Ref => Nat_Dates (I));
13                 ~~~~~~
14                 -- Returning reference object with a
15                 -- reference to the national day we
16                 -- found.
17             end if;
18         end loop;
19
20         return
21             National_Date_Reference'(Ref => null);
22         -- ~~~~~~
23         -- Returning reference object with a null
24         -- reference in case the country wasn't
```

(continues on next page)

(continued from previous page)

```

24     -- found. This will trigger an exception
25     -- if we try to dereference it.
26 end Find;
27
28 end National_Date_Containers;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.National_Dates
MD5: ec37ae93a7052c4bc731b2a7be0763ab

Package `National_Date_Containers` contains the `National_Dates` type, which is an array type for declaring containers that we use to store the national day information. We can also see the declaration of the `National_Date_Reference` type, which is the reference type returned by the `Find` function when looking for a specific country in the container.

Important

We're declaring the container type (`National_Dates`) as an array type just to simplify the code. In many cases, however, this approach isn't recommended! Instead, we should use a private type in order to encapsulate — and better protect — the information stored in the actual container.

Finally, let's see a test application that stores information for some countries into the `Nat_Dates` container and displays the information for a specific country:

Listing 1531: `show_national_dates.adb`

```

1  with National_Date_Info;
2  use  National_Date_Info;
3
4  with National_Date_Containers;
5  use  National_Date_Containers;
6
7  procedure Show_National_Dates is
8
9      Nat_Dates : constant National_Dates (1 .. 5) :=
10         (new National_Date'("USA",
11                               Time'(1776, 7, 4)),
12          new National_Date'("FRA",
13                               Time'(1789, 7, 14)),
14          new National_Date'("DEU",
15                               Time'(1990, 10, 3)),
16          new National_Date'("SPA",
17                               Time'(1492, 10, 12)),
18          new National_Date'("BRA",
19                               Time'(1822, 9, 7)));
20
21  begin
22      Find (Nat_Dates, "FRA").Show;
23      -- ^ implicit dereference
24  end Show_National_Dates;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.National_Dates
MD5: 771ecb91e8f890d4bb9b08115ae833f4

Runtime output

```
Country: FRA
Year:    1789
```

Here, we call the Find function to retrieve a reference object, whose reference (access value) has the national day information of France. We then implicitly dereference it to get the tagged object (of `National_Date` type) and display its information by calling the Show procedure.

Relevant topics

The `National_Date_Containers` package was implemented specifically as an accompanying package for the `National_Date_Info` package. It is possible, however, to generalize it, so that we can reuse the container for other record types. In fact, this is actually very straightforward:

Listing 1532: generic_containers.ads

```

1  generic
2    type T is private;
3    type T_Access is access T;
4    type T_Cmp is private;
5    with function Matches (E    : T_Access;
6                          Elem : T_Cmp)
7                          return Boolean;
8  package Generic_Containers is
9
10     type Ref_Type (Ref : access T) is
11       tagged limited null record
12       with Implicit_Dereference => Ref;
13
14     type Container is
15       array (Positive range <>) of
16         T_Access;
17
18     function Find (Cont : Container;
19                  Elem : T_Cmp)
20                  return Ref_Type;
21
22 end Generic_Containers;
```

Listing 1533: generic_containers.adb

```

1  package body Generic_Containers is
2
3     function Find (Cont : Container;
4                  Elem : T_Cmp)
5                  return Ref_Type is
6
7     begin
8       for I in Cont'Range loop
9         if Matches (Cont (I), Elem) then
10           return Ref_Type'(Ref => Cont (I));
11         end if;
12       end loop;
13
14       return Ref_Type'(Ref => null);
15     end Find;
16 end Generic_Containers;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.National_Dates
 MD5: 94c23a48131a47439b5b41e985c3d6c1

When comparing the **Generic**_Containers package to the `National_Date_Containers` package, we see that the main difference is the addition of the `Matches` function, which indicates whether the current element we're evaluating in the for-loop of the `Find` function is the one we're looking for.

In the main application, we can implement the `Matches` function and declare the `National_Date_Containers` package as an instance of the **Generic**_Containers package:

Listing 1534: `show_national_dates.adb`

```

1 with Generic_Containers;
2 with National_Date_Info; use National_Date_Info;
3
4 procedure Show_National_Dates is
5
6     function Matches_Country
7       (E : National_Date_Access;
8        Elem : Country_Code)
9       return Boolean is
10        (E.Country = Elem);
11
12     package National_Date_Containers is new
13       Generic_Containers
14       (T      => National_Date,
15        T_Access => National_Date_Access,
16        T_Cmp   => Country_Code,
17        Matches => Matches_Country);
18
19     use National_Date_Containers;
20
21     subtype National_Dates is Container;
22
23     Nat_Dates : constant
24       National_Dates (1 .. 5) :=
25       (new National_Date'("USA",
26                           Time'(1776, 7, 4)),
27        new National_Date'("FRA",
28                           Time'(1789, 7, 14)),
29        new National_Date'("DEU",
30                           Time'(1990, 10, 3)),
31        new National_Date'("SPA",
32                           Time'(1492, 10, 12)),
33        new National_Date'("BRA",
34                           Time'(1822, 9, 7)));
35
36 begin
37   Find (Nat_Dates, "FRA").Show;
38 end Show_National_Dates;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.User_Defined_References.National_Dates
 MD5: f4dac1fed69b9bccce5dccbf17844adc

Runtime output

Country: FRA
 Year: 1789

Here, we instantiate the **Generic_Containers** package with the `Matches_Country` function, which is an expression function that compares the country component of the current `National_Date` reference with the name of the country we desire to learn about.

This generalized approach is actually used for the standard containers from the Ada.Containers packages. For example, the `Ada.Containers.Vectors` is specified as follows:

```
with Ada.Iterator_Interfaces;

generic
  type Index_Type is range <>;
  type Element_Type is private;
  with function "=" (Left, Right : Element_Type)
    return Boolean is <>;
package Ada.Containers.Vectors
  with Preelaborate, Remote_Types,
    Nonblocking,
    Global => in out synchronized is

  -- OMITTED

  type Reference_Type
    (Element : not null access Element_Type) is
    private
      with Implicit_Dereference => Element,
        Nonblocking,
        Global => in out synchronized,
        Default_Initial_Condition =>
          (raise Program_Error);

  -- OMITTED

  function Reference
    (Container : aliased in out Vector;
     Index      : in Index_Type)
    return Reference_Type
    with Pre => Index in
      First_Index (Container) ..
      Last_Index (Container)
    or else raise
      Constraint_Error,

    Post =>
      Tampering_With_Cursors_Prohibited
        (Container),
    Nonblocking,
    Global => null,
    Use_Formal => null;

  -- OMITTED

  function Reference
    (Container : aliased in out Vector;
     Position  : in Cursor)
    return Reference_Type
    with Pre => (Position /= No_Element
      or else raise
        Constraint_Error)
    and then
      (Has_Element
        (Container, Position)
      or else raise
        Program_Error),
```

```

        Post    =>
            Tampering_With_Cursors_Prohibited
                (Container),
            Nonblocking,
            Global => null,
            Use_Formal => null;

        -- OMITTED

    end Ada.Containers.Vectors;

```

(Note that most parts of the Vectors package were omitted for clarity. Please refer to the Ada Reference Manual for the complete package specification.)

Here, we see that the `Implicit_Dereference` aspect is used in the declaration of **Reference_Type**, which is the reference type returned by the Reference functions for an index or a cursor.

Also, note that the Vectors package has a formal equality function (`=`) instead of the `Matches` function we were using in our **Generic_Containers** package. The purpose of the formal function, however, is basically the same.

In the Ada Reference Manual

- A.18.2 The Generic Package `Containers.Vectors`²⁸⁵

28.2.8 Anonymous Access Types and Accessibility Rules

In general, the *accessibility rules* (page 934) we've seen earlier also apply to anonymous access types. However, there are some subtle differences, which we discuss in this section.

Let's adapt the *code example from that section* (page 935) to make use of anonymous access types:

Listing 1535: `library_level.ads`

```

1 package Library_Level is
2
3     L0_A0 : access Integer;
4
5     L0_Var : aliased Integer;
6
7 end Library_Level;

```

Listing 1536: `show_library_level.adb`

```

1 with Library_Level; use Library_Level;
2
3 procedure Show_Library_Level is
4     L1_Var : aliased Integer;
5
6     L1_A0 : access Integer;
7
8     procedure Test is
9         L2_A0 : access Integer;
10
11         L2_Var : aliased Integer;
12     begin

```

(continues on next page)

²⁸⁵ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

(continued from previous page)

```

13     L1_A0 := L2_Var'Access;
14     --      ^^^^^^
15     --      ILLEGAL: L2 object to
16     --      L1 access object
17
18     L2_A0 := L2_Var'Access;
19     --      ^^^^^^
20     --      LEGAL: L2 object to
21     --      L2 access object
22 end Test;
23
24 begin
25     L0_A0 := new Integer'(22);
26     --      ^^^^^^^^^^
27     --      LEGAL: L0 object to
28     --      L0 access object
29
30     L0_A0 := L1_Var'Access;
31     --      ^^^^^^
32     --      ILLEGAL: L1 object to
33     --      L0 access object
34
35     L1_A0 := L0_Var'Access;
36     --      ^^^^^^
37     --      LEGAL: L0 object to
38     --      L1 access object
39
40     L1_A0 := L1_Var'Access;
41     --      ^^^^^^
42     --      LEGAL: L1 object to
43     --      L1 access object
44
45     L0_A0 := L1_A0; -- legal!!
46     --      ^^^^^^
47     --      LEGAL: L1 access object to
48     --      L0 access object
49
50     --      ILLEGAL: L1 object
51     --      (L1_A0 = L1_Var'Access)
52     --      to
53     --      L0 access object
54
55     --      This is actually OK at compile time,
56     --      but the accessibility check fails at
57     --      runtime.
58
59     Test;
60 end Show_Library_Level;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↳ Accessibility_Levels_Rules_Introduction.Accessibility_Library_Level
 MD5: 255bdecebd735408db082edd583a0c

Build output

```

show_library_level.adb:13:16: error: non-local pointer cannot point to local object
show_library_level.adb:30:13: error: non-local pointer cannot point to local object
gprbuild: *** compilation phase failed

```

As we see in the code, in general, most accessibility rules are the same as the ones we've

discussed when using named access types. For example, an assignment such as `L0_A0 := L1_Var'Access` is illegal because we're trying to assign to an access object of less deep level.

However, assignment such as `L0_A0 := L1_A0` are possible now: we don't get a type mismatch — as we did with named access types — because both objects are of anonymous access types. Note that the accessibility level cannot be determined at compile time: `L1_A0` can hold an access value at library level (which would make the assignment legal) or at a deeper level. Therefore, the compiler introduces an accessibility check here.

However, the accessibility check used in `L0_A0 := L1_A0` fails at runtime because the corresponding access value (`L1_Var'Access`) is of a deeper level than `L0_A0`, which is illegal. (If you comment out the `L1_A0 := L1_Var'Access` assignment prior to the `L0_A0 := L1_A0` assignment, this accessibility check doesn't fail anymore.)

Conversions between Anonymous and Named Access Types

In the previous sections, we've discussed accessibility rules for named and anonymous access types separately. In this section, we see that the same accessibility rules apply when mixing both flavors together and converting objects of anonymous to named access types.

Let's adapt parts of the previous [code example](#) (page 935) and add anonymous access types to it:

Listing 1537: library_level.ads

```

1 package Library_Level is
2
3     type L0_Integer_Access is
4         access all Integer;
5
6     L0_Var : aliased Integer;
7
8     L0_IA  : L0_Integer_Access;
9     L0_A0  : access Integer;
10
11 end Library_Level;
```

Listing 1538: show_library_level.adb

```

1 with Library_Level; use Library_Level;
2
3 procedure Show_Library_Level is
4     type L1_Integer_Access is
5         access all Integer;
6
7     L1_IA  : L1_Integer_Access;
8     L1_A0  : access Integer;
9
10    L1_Var : aliased Integer;
11
12 begin
13     -----
14     -- From named type to anonymous type
15     -----
16
17     L0_IA := new Integer'(22);
18     L1_IA := new Integer'(42);
19
20     L0_A0 := L0_IA;
21     --
```

(continues on next page)

(continued from previous page)

```

22      --      LEGAL: assignment from
23      --      L0 access object (named type)
24      --      to
25      --      L0 access object
26      --      (anonymous type)
27
28  L0_A0 := L1_IA;
29      --      ^^^^^
30      --      ILLEGAL: assignment from
31      --      L1 access object (named type)
32      --      to
33      --      L0 access object
34      --      (anonymous type)
35
36  L1_A0 := L0_IA;
37      --      ^^^^^
38      --      LEGAL: assignment from
39      --      L0 access object (named type)
40      --      to
41      --      L1 access object
42      --      (anonymous type)
43
44  L1_A0 := L1_IA;
45      --      ^^^^^
46      --      LEGAL: assignment from
47      --      L1 access object (named type)
48      --      to
49      --      L1 access object
50      --      (anonymous type)
51
52  -----
53  --  From anonymous type to named type
54  -----
55
56  L0_A0 := L0_Var'Access;
57  L1_A0 := L1_Var'Access;
58
59  L0_IA := L0_Integer_Access (L0_A0);
60      --      ^^^^^^^^^^^^^^^^^
61      --      LEGAL: conversion / assignment from
62      --      L0 access object
63      --      (anonymous type)
64      --      to
65      --      L0 access object (named type)
66
67  L0_IA := L0_Integer_Access (L1_A0);
68      --      ^^^^^^^^^^^^^^^^^
69      --      ILLEGAL: conversion / assignment from
70      --      L1 access object
71      --      (anonymous type)
72      --      to
73      --      L0 access object (named type)
74      --      (accessibility check fails)
75
76  L1_IA := L1_Integer_Access (L0_A0);
77      --      ^^^^^^^^^^^^^^^^^
78      --      LEGAL: conversion / assignment from
79      --      L0 access object
80      --      (anonymous type)
81      --      to
82      --      L1 access object (named type)

```

(continues on next page)

(continued from previous page)

```

83
84   L1_IA := L1_Integer_Access (L1_A0);
85   -- ~~~~~
86   --     LEGAL: conversion / assignment from
87   --           L1 access object
88   --           (anonymous type)
89   --           to
90   --           L1 access object (named type)
91 end Show_Library_Level;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↳ Accessibility_Levels_Rules_Introduction.Accessibility_Named_Anonymous_Access_
 ↳ Type_Conversions
 MD5: a2e73bb0ed543bc4973850c80f951039

Build output

```

show_library_level.adb:28:13: error: cannot convert local pointer to non-local_
↳ access type
gprbuild: *** compilation phase failed
```

As we can see in this code example, mixing access objects of named and anonymous access types doesn't change the accessibility rules. Again, the rules are only violated when the target object in the assignment is *less* deep. This is the case in the `L0_A0 := L1_IA` and the `L0_IA := L0_Integer_Access (L1_A0)` assignments. Otherwise, mixing those access objects doesn't impose additional hurdles.

Accessibility rules on access parameters

In the previous chapter, we saw that the accessibility rules also apply to *access values as subprogram parameters* (page 938). In the case of access parameters, the rules are a bit less strict (as you may generally expect for anonymous access types), and the accessibility rules are checked at runtime. This allows use to use access values that would be illegal in the case of named access types because of their accessibility levels.

Let's adapt a previous code example to make use of access parameters:

Listing 1539: names.ads

```

1 package Names is
2
3   procedure Show (N : access constant String);
4
5 end Names;
```

Listing 1540: names.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 -- with Ada.Characters.Handling;
4 -- use Ada.Characters.Handling;
5
6 package body Names is
7
8   procedure Show (N : access constant String) is
9   begin
10     -- for I in N'Range loop
11     --   N (I) := To_Lower (N (I));
```

(continues on next page)

(continued from previous page)

```
12     -- end loop;
13     Put_Line ("Name: " & N.all);
14 end Show;
15
16 end Names;
```

Listing 1541: show_names.adb

```
1 with Names; use Names;
2
3 procedure Show_Names is
4   S : aliased String := "John";
5 begin
6   Show (S'Access);
7 end Show_Names;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Access_Types.Accessibility_Levels_Rules_Introduction.Accessibility_Checks_Parameters
MD5: aa930ba9be3264d01eb9115d27b884eb

Runtime output

Name: John

As we've seen in the previous chapter, compilation fails when we use named access types in this code example. In the case of access parameters, using `S'Access` doesn't make the compilation fail, nor does the accessibility check fail at runtime because `S` is still in scope when we call the `Show` procedure.

28.2.9 Anonymous Access-To-Subprograms

In the previous chapter, we talked about *named access-to-subprogram types* (page 964). Now, we'll see that the anonymous version of those types isn't much different from the named version.

Let's start our discussion by declaring a subprogram parameter using an anonymous access-to-procedure type:

Listing 1542: anonymous_access_to_subprogram.ads

```
1 package Anonymous_Access_To_Subprogram is
2
3   procedure Proc
4     (P : access procedure (I : in out Integer));
5
6 end Anonymous_Access_To_Subprogram;
```

Listing 1543: anonymous_access_to_subprogram.adb

```
1 package body Anonymous_Access_To_Subprogram is
2
3   procedure Proc
4     (P : access procedure (I : in out Integer))
5   is
6     I : Integer := 0;
7   begin
8     P (I);
```

(continues on next page)

(continued from previous page)

```

9   end Proc;
10
11  end Anonymous_Access_To_Subprogram;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
 MD5: 2cbe76d7e23905d575bd27e29d5e3175

In this example, we use the anonymous **access procedure** (**I : in out Integer**) type as a parameter of the Proc procedure. Note that we need an identifier in the declaration: we cannot leave I out and write **access procedure** (**in out Integer**).

Before we look at a test application that makes use of the Anonymous_Access_To_Subprogram package, let's implement two simple procedures that we'll use later on:

Listing 1544: add_ten.ads

```

1  procedure Add_Ten (I : in out Integer);

```

Listing 1545: add_ten.adb

```

1  procedure Add_Ten (I : in out Integer) is
2  begin
3      I := I + 10;
4  end Add_Ten;

```

Listing 1546: add_twenty.ads

```

1  procedure Add_Twenty (I : in out Integer);

```

Listing 1547: add_twenty.adb

```

1  procedure Add_Twenty (I : in out Integer) is
2  begin
3      I := I + 20;
4  end Add_Twenty;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
 MD5: 50eaeaf27caaa9618b35ecdf8acc11fe

Finally, this is our test application:

Listing 1548: show_anonymous_access_to_subprograms.adb

```

1  with Anonymous_Access_To_Subprogram;
2  use  Anonymous_Access_To_Subprogram;
3
4  with Add_Ten;
5
6  procedure Show_Anonymous_Access_To_Subprograms is
7  begin
8      Proc (Add_Ten'Access);
9      --      ^ Getting access to Add_Ten
10     --      procedure and passing it

```

(continues on next page)

(continued from previous page)

```

11      --      to Proc
12 end Show_Anonymous_Access_To_Subprograms;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 13143ccf9620d26031484ba160a58fe1

Here, we get access to the `Add_Ten` procedure and pass it to the `Proc` procedure. Note that this implementation is not different from the *example for named access-to-subprogram types* (page 966). In fact, in terms of usage, anonymous access-to-subprogram types are very similar to named access-to-subprogram types. The major differences can be found in the corresponding *accessibility rules* (page 1053).

In the Ada Reference Manual

- 3.10 Access Types²⁸⁶

Examples of anonymous access-to-subprogram usage

In the section about *named access-to-subprogram types* (page 964), we've seen a couple of different usages for those types. In all those examples we discussed, we could instead have used anonymous access-to-subprogram types. Let's see a code example that illustrates that:

Listing 1549: all_anonymous_access_to_subprogram.ads

```

1 package All_Anonymous_Access_To_Subprogram is
2
3     --
4     --  Anonymous access-to-subprogram as
5     --  subprogram parameter:
6     --
7     procedure Proc
8         (P : access procedure (I : in out Integer));
9
10    --
11    --  Anonymous access-to-subprogram in
12    --  array type declaration:
13    --
14    type Access_To_Procedure_Array is
15        array (Positive range <>) of
16            access procedure (I : in out Integer);
17
18    protected type Protected_Integer is
19
20        procedure Mult_Ten;
21
22        procedure Mult_Twenty;
23
24    private
25        I : Integer := 1;
26    end Protected_Integer;
27
28    --
```

(continues on next page)

²⁸⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

(continued from previous page)

```

29  -- Anonymous access-to-subprogram as
30  -- component of a record type.
31  --
32  type Rec_Access_To_Procedure is record
33      AP : access procedure (I : in out Integer);
34  end record;
35
36  --
37  -- Anonymous access-to-subprogram as
38  -- discriminant:
39  --
40  type Rec_Access_To_Procedure_Discriminant
41      (AP : access procedure
42       (I : in out Integer)) is
43  record
44      I : Integer := 0;
45  end record;
46
47  procedure Process
48      (R : in out
49       Rec_Access_To_Procedure_Discriminant);
50
51  generic
52      type T is private;
53
54      --
55      -- Anonymous access-to-subprogram as
56      -- formal parameter:
57      --
58      Proc_T : access procedure
59          (Element : in out T);
60  procedure Gen_Process (Element : in out T);
61
62  end All_Anonymous_Access_To_Subprogram;

```

Listing 1550: all_anonymous_access_to_subprogram.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body All_Anonymous_Access_To_Subprogram is
4
5      procedure Proc
6          (P : access procedure (I : in out Integer))
7      is
8          I : Integer := 0;
9      begin
10         Put_Line
11             ("Calling procedure for Proc...");
12         P (I);
13         Put_Line ("Finished.");
14     end Proc;
15
16     procedure Process
17         (R : in out
18          Rec_Access_To_Procedure_Discriminant)
19     is
20     begin
21         Put_Line
22             ("Calling procedure for"
23              & " Rec_Access_To_Procedure_Discriminant"
24              & " type...");

```

(continues on next page)

(continued from previous page)

```

25     R.AP (R.I);
26     Put_Line ("Finished.");
27 end Process;
28
29 procedure Gen_Process (Element : in out T) is
30 begin
31     Put_Line
32     ("Calling procedure for Gen_Process...");
33     Proc_T (Element);
34     Put_Line ("Finished.");
35 end Gen_Process;
36
37 protected body Protected_Integer is
38
39     procedure Mult_Ten is
40     begin
41         I := I * 10;
42     end Mult_Ten;
43
44     procedure Mult_Twenty is
45     begin
46         I := I * 20;
47     end Mult_Twenty;
48
49 end Protected_Integer;
50
51 end All_Anonymous_Access_To_Subprogram;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
MD5: 628dcfdc5fe9b712f33fa044057093c2

In the All_Anonymous_Access_To_Subprogram package, we see examples of anonymous access-to-subprogram types:

- as a subprogram parameter;
- in an array type declaration;
- as a component of a record type;
- as a record type discriminant;
- as a formal parameter of a generic procedure.

Let's implement a test application that makes use of this package:

Listing 1551: show_anonymous_access_to_subprograms.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Add_Ten;
4  with Add_Twenty;
5
6  with All_Anonymous_Access_To_Subprogram;
7  use All_Anonymous_Access_To_Subprogram;
8
9  procedure Show_Anonymous_Access_To_Subprograms is
10     --
11     --   Anonymous access-to-subprogram as
12     --   an object:

```

(continues on next page)

(continued from previous page)

```

13  --
14  P  : access procedure (I : in out Integer);
15
16  --
17  --  Array of anonymous access-to-subprogram
18  --  components
19  --
20  PA : constant
21      Access_To_Procedure_Array (1 .. 2) :=
22      (Add_Ten'Access,
23       Add_Twenty'Access);
24
25  --
26  --  Anonymous array of anonymous
27  --  access-to-subprogram components:
28  --
29  PAA : constant
30      array (1 .. 2) of access
31      procedure (I : in out Integer) :=
32      (Add_Ten'Access,
33       Add_Twenty'Access);
34
35  --
36  --  Record with anonymous
37  --  access-to-subprogram components:
38  --
39  RA : constant Rec_Access_To_Procedure :=
40      (AP => Add_Ten'Access);
41
42  --
43  --  Record with anonymous
44  --  access-to-subprogram discriminant:
45  --
46  RD : Rec_Access_To_Procedure_Discriminant
47      (AP => Add_Twenty'Access) :=
48      (AP => Add_Twenty'Access, I => 0);
49
50  --
51  --  Generic procedure with formal anonymous
52  --  access-to-subprogram:
53  --
54  procedure Process_Integer is new
55      Gen_Process (T      => Integer,
56                  Proc_T => Add_Twenty'Access);
57
58  --
59  --  Object (APP) of anonymous
60  --  access-to-protected-subprogram:
61  --
62  PI : Protected_Integer;
63  APP : constant access protected procedure :=
64      PI.Mult_Ten'Access;
65
66  Some_Int : Integer := 0;
67  begin
68      Put_Line ("Some_Int: " & Some_Int'Image);
69
70  --
71  --  Using object of
72  --  anonymous access-to-subprogram type:
73  --

```

(continues on next page)

(continued from previous page)

```

74  P := Add_Ten'Access;
75  Proc (P);
76  P (Some_Int);
77
78  P := Add_Twenty'Access;
79  Proc (P);
80  P (Some_Int);
81
82  Put_Line ("Some_Int: " & Some_Int'Image);
83
84  --
85  --  Using array with component of
86  --  anonymous access-to-subprogram type:
87  --
88  Put_Line
89    ("Calling procedure from PA array...");
90
91  for I in PA'Range loop
92    PA (I) (Some_Int);
93    Put_Line ("Some_Int: " & Some_Int'Image);
94  end loop;
95
96  Put_Line ("Finished.");
97
98  Put_Line
99    ("Calling procedure from PAA array...");
100
101  for I in PAA'Range loop
102    PAA (I) (Some_Int);
103    Put_Line ("Some_Int: " & Some_Int'Image);
104  end loop;
105
106  Put_Line ("Finished.");
107
108  Put_Line ("Some_Int: " & Some_Int'Image);
109
110  --
111  --  Using record with component of
112  --  anonymous access-to-subprogram type:
113  --
114  RA.AP (Some_Int);
115  Put_Line ("Some_Int: " & Some_Int'Image);
116
117  --
118  --  Using record with discriminant of
119  --  anonymous access-to-subprogram type:
120  --
121  Process (RD);
122  Put_Line ("RD.I: " & RD.I'Image);
123
124  --
125  --  Using procedure instantiated with
126  --  formal anonymous access-to-subprogram:
127  --
128  Process_Integer (Some_Int);
129  Put_Line ("Some_Int: " & Some_Int'Image);
130
131  --
132  --  Using object of anonymous
133  --  access-to-protected-subprogram type:
134  --

```

(continues on next page)

(continued from previous page)

```

135     APP.all;
136 end Show_Anonymous_Access_To_Subprograms;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Anonymous_Access_To_Subprogram_Example
 MD5: ec770c17e880a98fd2e9ab0110d4a858

Runtime output

```

Some_Int: 0
Calling procedure for Proc...
Finished.
Calling procedure for Proc...
Finished.
Some_Int: 30
Calling procedure from PA array...
Some_Int: 40
Some_Int: 60
Finished.
Calling procedure from PAA array...
Some_Int: 70
Some_Int: 90
Finished.
Some_Int: 90
Some_Int: 100
Calling procedure for Rec_Access_To_Procedure_Discriminant type...
Finished.
RD.I: 20
Calling procedure for Gen_Process...
Finished.
Some_Int: 120

```

In the `Show_Anonymous_Access_To_Subprograms` procedure, we see examples of anonymous access-to-subprogram types in:

- in objects (P) and (APP);
- in arrays (PA and PAA);
- in records (RA and RD);
- in the binding to a formal parameter (Proc_T) of an instantiated procedure (Process_Integer);
- as a parameter of a procedure (Proc).

Because we already discussed all these usages in the section about *named access-to-subprogram types* (page 964), we won't repeat this discussion here. If anything in this code example is still unclear to you, make sure to revisit that section from the previous chapter.

Application of anonymous access-to-subprogram types

In general, there isn't much that speaks against using anonymous access-to-subprogram types. We can say, for example, that they're much more useful than *anonymous access-to-objects types* (page 1002), which have *many drawbacks* (page 1004) — as we discussed earlier.

There isn't much to be concerned when using anonymous access-to-subprogram types. For example, we cannot allocate or deallocate a subprogram. As a consequence, we won't

have storage management issues affecting these types because the access to those subprograms will always be available and no memory leak can occur.

Also, anonymous access-to-subprogram types can be easier to use than named access-to-subprogram types because of their less strict *accessibility rules* (page 1053). Some of the accessibility issues we might encounter when using named access-to-subprogram types can be solved by declaring them as anonymous types. (We discuss the accessibility rules of anonymous access-to-subprogram types in the next section.)

Readability

Note that readability suffers if you use a *cascade* of anonymous access-to-subprograms. For example:

Listing 1552: readability_issue.ads

```
1 package Readability_Issue is
2
3     function F
4         return access
5         function (A : Integer)
6             return access
7             function (B : Float)
8                 return Integer;
9
10 end Readability_Issue;
```

Listing 1553: readability_issue-functions.ads

```
1 package Readability_Issue.Functions is
2
3     function To_Integer (V : Float)
4         return Integer is
5         (Integer (V));
6
7     function Select_Conversion
8         (A : Integer)
9         return access
10         function (B : Float)
11             return Integer is
12         (To_Integer'Access);
13
14 end Readability_Issue.Functions;
```

Listing 1554: readability_issue.adb

```
1 with Readability_Issue.Functions;
2 use Readability_Issue.Functions;
3
4 package body Readability_Issue is
5
6     function F
7         return access
8         function (A : Integer)
9             return access
10             function (B : Float)
11                 return Integer is
12         (Select_Conversion'Access);
13
14 end Readability_Issue;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Readability_Issue
 MD5: 9e2ac58942c97b44c0d847c28e39bd11

In this example, the definition of F might compile fine, but it's simply too long to be readable. Not only that: we need to carry this *chain* to other functions as well — such as the `Select_Conversion` function above. Also, using these functions in an application is not straightforward:

Listing 1555: show_readability_issue.adb

```

1 with Readability_Issue;
2 use Readability_Issue;
3
4 procedure Show_Readability_Issue is
5   F1 : access
6       function (A : Integer)
7           return access
8               function (B : Float)
9                   return Integer
10          := F;
11   F2 : access function (B : Float)
12       return Integer
13      := F1 (2);
14   I : Integer := F2 (0.1);
15 begin
16   I := F1 (2) (0.1);
17 end Show_Readability_Issue;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.Anonymous_Access_To_Subprograms.Readability_Issue
 MD5: 80267b1d673663e3cacba0c4978e6abf

Therefore, our recommendation is to avoid this kind of *access cascading* by carefully designing your application. In general, you won't need that.

28.2.10 Accessibility Rules and Anonymous Access-To-Subprograms

In principle, the *accessibility rules for anonymous access types* (page 1039) that we've seen before apply to anonymous access-to-subprograms as well. Also, we had a discussion about *accessibility rules and access-to-subprograms* (page 989) in the previous chapter. In this section, we review some of the rules that we already know and discuss how they relate to anonymous access-to-subprograms.

In the Ada Reference Manual

- 3.10 Access Types²⁸⁷

Named vs. anonymous access-to-subprograms

Let's see an example of a named access-to-subprogram type:

²⁸⁷ <http://www.ada-auth.org/standards/22rm/html/RM-3-10.html>

Listing 1556: show_access_to_subprogram_error.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Access_To_Subprogram_Error is
4
5      type PI is access
6          procedure (I : in out Integer);
7
8      P : PI;
9
10     I : Integer := 0;
11 begin
12     declare
13         procedure Add_One (I : in out Integer) is
14             begin
15                 I := I + 1;
16             end Add_One;
17         begin
18             P := Add_One'Access;
19         end;
20 end Show_Access_To_Subprogram_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪ Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Named
 MD5: 41c36426112e799210b7704dd43b6217

Build output

```

show_access_to_subprogram_error.adb:18:12: error: subprogram must not be deeper_
↪ than access type
gprbuild: *** compilation phase failed
```

In this example, we get a compilation error because the lifetime of the `Add_One` procedure is shorter than the access type `PI`.

In contrast, using an anonymous access-to-subprogram type eliminates the compilation error, i.e. the assignment `P := Add_One'Access` becomes legal:

Listing 1557: show_access_to_subprogram_error.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Access_To_Subprogram_Error is
4      P : access procedure (I : in out Integer);
5
6      I : Integer := 0;
7  begin
8      declare
9          procedure Add_One (I : in out Integer) is
10              begin
11                  I := I + 1;
12              end Add_One;
13          begin
14              P := Add_One'Access;
15              --  RUNTIME ERROR: Add_One is out-of-scope
16              --                    after this line.
17          end;
18 end Show_Access_To_Subprogram_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪ Accessibility_Rules_Anonymous_Access_To_Subprograms.Simple_Example_Anonymous
 MD5: a5eeb4a716b4f6a932dd74c580a07b66

Runtime output

```
raised PROGRAM_ERROR : show_access_to_subprogram_error.adb:14 accessibility check_
↪ failed
```

In this case, the compiler introduces an accessibility check, which fails at runtime because the lifetime of `Add_One` is shorter than the lifetime of the access object `P`.

Named vs. anonymous access-to-subprograms as parameters

Using anonymous access-to-subprograms as parameters allows us to pass subprograms at any level. For certain applications, the restrictions that are applied to named access types might be too strict, so using anonymous access-to-subprograms might be a good way to circumvent those restrictions. They also allow the component developer to be independent of the clients' specific access types.

Note that the increased flexibility for anonymous access-to-subprograms means that some of the checks that are performed at compile time for named access-to-subprograms are done at runtime for anonymous access-to-subprograms.

Named access-to-subprograms as a parameter

Let's see an example using a named access-to-procedure type:

Listing 1558: `access_to_subprogram_types.ads`

```
1 package Access_To_Subprogram_Types is
2
3   type Integer_Array is
4     array (Positive range <>) of Integer;
5
6   type Process_Procedure is
7     access
8     procedure (Arr : in out Integer_Array);
9
10  procedure Process
11    (Arr : in out Integer_Array;
12     P   : Process_Procedure);
13
14 end Access_To_Subprogram_Types;
```

Listing 1559: `access_to_subprogram_types.adb`

```
1 package body Access_To_Subprogram_Types is
2
3   procedure Process
4     (Arr : in out Integer_Array;
5     P   : Process_Procedure) is
6   begin
7     P (Arr);
8   end Process;
9
10 end Access_To_Subprogram_Types;
```

Listing 1560: show_access_to_subprogram_error.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Access_To_Subprogram_Types;
4  use  Access_To_Subprogram_Types;
5
6  procedure Show_Access_To_Subprogram_Error is
7
8      procedure Add_One
9          (Arr : in out Integer_Array) is
10         begin
11             for E of Arr loop
12                 E := E + 1;
13             end loop;
14         end Add_One;
15
16         procedure Display
17             (Arr : in out Integer_Array) is
18             begin
19                 for I in Arr'Range loop
20                     Put_Line ("Arr (" &
21                             Integer'Image (I)
22                             & "): "
23                             & Integer'Image (Arr (I)));
24                 end loop;
25             end Display;
26
27         Arr : Integer_Array (1 .. 3) := (1, 2, 3);
28     begin
29         Process (Arr, Display'Access);
30
31         Put_Line ("Add_One...");
32         Process (Arr, Add_One'Access);
33
34         Process (Arr, Display'Access);
35     end Show_Access_To_Subprogram_Error;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
 ↪ Accessibility_Rules_Anonymous_Access_To_Subprograms.Access_To_Subprogram_
 ↪ Parameter_Named
 MD5: 76b70b52a0374fe0fd398024fe869876

Build output

```

show_access_to_subprogram_error.adb:29:18: error: subprogram must not be deeper
↪ than access type
show_access_to_subprogram_error.adb:32:18: error: subprogram must not be deeper
↪ than access type
show_access_to_subprogram_error.adb:34:18: error: subprogram must not be deeper
↪ than access type
gprbuild: *** compilation phase failed

```

In this example, we declare the `Process_Procedure` type in the `Access_To_Subprogram_Types` package and use it in the `Process` procedure, which we call in the `Show_Access_To_Subprogram_Error` procedure. The accessibility rules trigger a compilation error because the accesses (`Add_One'Access` and `Display'Access`) are at a deeper level than the access-to-procedure type (`Process_Procedure`).

As we know already, there's no `Unchecked_Access` attribute that we could use here. An

easy way to make this code compile could be to move `Add_One` and `Display` to the library level.

Anonymous access-to-subprograms as a parameter

To circumvent the compilation error, we could also use anonymous access-to-subprograms instead:

Listing 1561: `access_to_subprogram_types.ads`

```

1 package Access_To_Subprogram_Types is
2
3     type Integer_Array is
4         array (Positive range <>) of Integer;
5
6     procedure Process
7         (Arr : in out Integer_Array;
8          P   : access procedure
9              (Arr : in out Integer_Array));
10
11 end Access_To_Subprogram_Types;
```

Listing 1562: `access_to_subprogram_types.adb`

```

1 package body Access_To_Subprogram_Types is
2
3     procedure Process
4         (Arr : in out Integer_Array;
5          P   : access procedure
6              (Arr : in out Integer_Array)) is
7     begin
8         P (Arr);
9     end Process;
10
11 end Access_To_Subprogram_Types;
```

Listing 1563: `show_access_to_subprogram_error.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Access_To_Subprogram_Types;
4 use Access_To_Subprogram_Types;
5
6 procedure Show_Access_To_Subprogram_Error is
7
8     procedure Add_One
9         (Arr : in out Integer_Array) is
10    begin
11        for E of Arr loop
12            E := E + 1;
13        end loop;
14    end Add_One;
15
16    procedure Display
17        (Arr : in out Integer_Array) is
18    begin
19        for I in Arr'Range loop
20            Put_Line ("Arr (" &
21                     Integer'Image (I)
22                     & "): "
23                     & Integer'Image (Arr (I)));
```

(continues on next page)

(continued from previous page)

```
24     end loop;
25 end Display;
26
27 Arr : Integer_Array (1 .. 3) := (1, 2, 3);
28 begin
29     Process (Arr, Display'Access);
30
31     Put_Line ("Add_One...");
32     Process (Arr, Add_One'Access);
33
34     Process (Arr, Display'Access);
35 end Show_Access_To_Subprogram_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳ Accessibility_Rules_Anonymous_Access_To_Subprograms.Access_To_Subprogram_
↳ Parameter_Anonymous
MD5: a500e0a864f0adadc1d6823c1f50bd64

Runtime output

```
Arr ( 1): 1
Arr ( 2): 2
Arr ( 3): 3
Add_One...
Arr ( 1): 2
Arr ( 2): 3
Arr ( 3): 4
```

Now, the code is accepted by the compiler because anonymous access-to-subprograms used as parameters allow passing of subprograms at any level. Also, we don't see a runtime exception because the subprograms are still *accessible* when we call `Process`.

Iterator

A typical example that illustrates well the necessity of using anonymous access-to-subprograms is that of a container iterator. In fact, many of the standard Ada containers — the child packages of `Ada.Containers` — make use of anonymous access-to-subprograms for their `Iterate` subprograms.

In the Ada Reference Manual

- [A.18.2 The Package Containers.Vectors](#)²⁸⁸
- [A.18.4 Maps](#)²⁸⁹
- [A.18.7 Sets](#)²⁹⁰

Using named access-to-subprograms

Let's start with a simplified container type (`Data_Container`) using a named access-to-subprogram type (`Process_Element`) for iteration:

²⁸⁸ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-2.html>

²⁸⁹ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-4.html>

²⁹⁰ <http://www.ada-auth.org/standards/22rm/html/RM-A-18-7.html>

Listing 1564: data_processing.ads

```

1 generic
2   type Element is private;
3 package Data_Processing is
4
5   type Data_Container (Last : Positive) is
6     private;
7
8   Data_Container_Full : exception;
9
10  procedure Append (D : in out Data_Container;
11                  E :      Element);
12
13  type Process_Element is
14    not null access procedure (E : Element);
15
16  procedure Iterate
17    (D      : Data_Container;
18     Proc   : Process_Element);
19
20 private
21
22  type Data_Container_Storage is
23    array (Positive range <>) of Element;
24
25  type Data_Container (Last : Positive) is
26    record
27      S      : Data_Container_Storage (1 .. Last);
28      Curr : Natural := 0;
29    end record;
30
31 end Data_Processing;
```

Listing 1565: data_processing.adb

```

1 package body Data_Processing is
2
3   procedure Append (D : in out Data_Container;
4                   E :      Element) is
5   begin
6     if D.Curr < D.S'Last then
7       D.Curr := D.Curr + 1;
8       D.S (D.Curr) := E;
9     else
10      raise Data_Container_Full;
11      -- NOTE: This is just a dummy
12      --       implementation. A better
13      --       strategy is to add actual error
14      --       handling when the container is
15      --       full.
16    end if;
17  end Append;
18
19  procedure Iterate
20    (D      : Data_Container;
21     Proc   : Process_Element) is
22  begin
23    for I in D.S'First .. D.Curr loop
24      Proc (D.S (I));
25    end loop;
```

(continues on next page)

(continued from previous page)

```
26     end Iterate;
27
28 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Named
MD5: e48e8200e571b62d027753ee96c47fcb

In this example, we declare the `Process_Element` type in the generic `Data_Processing` package, and we use it in the `Iterate` procedure. We then instantiate this package as `Float_Data_Processing`, and we use it in the `Show_Access_To_Subprograms` procedure:

Listing 1566: float_data_processing.ads

```
1 with Data_Processing;
2
3 package Float_Data_Processing is
4     new Data_Processing (Element => Float);
```

Listing 1567: show_access_to_subprograms.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Data_Processing;
4 use Float_Data_Processing;
5
6 procedure Show_Access_To_Subprograms is
7
8     procedure Display (F : Float) is
9     begin
10         Put_Line ("F :" & Float'Image (F));
11     end Display;
12
13     D : Data_Container (5);
14 begin
15     Append (D, 1.0);
16     Append (D, 2.0);
17     Append (D, 3.0);
18
19     Iterate (D, Display'Access);
20 end Show_Access_To_Subprograms;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Named
MD5: 64ee435aac5f2817b7d9cecf538a1e4c

Build output

```
show_access_to_subprograms.adb:19:17: error: subprogram must not be deeper than_
↳access type
gprbuild: *** compilation phase failed
```

Using `Display'Access` in the call to `Iterate` triggers a compilation error because its lifetime is shorter than the lifetime of the `Process_Element` type.

Using anonymous access-to-subprograms

Now, let's use an anonymous access-to-subprogram type in the Iterate procedure:

Listing 1568: data_processing.ads

```

1  generic
2      type Element is private;
3  package Data_Processing is
4
5      type Data_Container (Last : Positive) is
6          private;
7
8      Data_Container_Full : exception;
9
10     procedure Append (D : in out Data_Container;
11                      E :      Element);
12
13     procedure Iterate
14         (D      : Data_Container;
15          Proc : not null access
16              procedure (E : Element));
17
18 private
19
20     type Data_Container_Storage is
21         array (Positive range <>) of Element;
22
23     type Data_Container (Last : Positive) is
24         record
25             S      : Data_Container_Storage (1 .. Last);
26             Curr : Natural := 0;
27         end record;
28
29 end Data_Processing;
```

Listing 1569: data_processing.adb

```

1  package body Data_Processing is
2
3      procedure Append (D : in out Data_Container;
4                      E :      Element) is
5      begin
6          if D.Curr < D.S'Last then
7              D.Curr := D.Curr + 1;
8              D.S (D.Curr) := E;
9          else
10             raise Data_Container_Full;
11             -- NOTE: This is just a dummy
12             --       implementation. A better
13             --       strategy is to add actual error
14             --       handling when the container is
15             --       full.
16         end if;
17     end Append;
18
19     procedure Iterate
20         (D      : Data_Container;
21          Proc : not null access
22              procedure (E : Element)) is
23     begin
24         for I in D.S'First .. D.Curr loop
```

(continues on next page)

(continued from previous page)

```
25         Proc (D.S (I));
26     end loop;
27 end Iterate;
28
29 end Data_Processing;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳ Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: fa56595ef1734f2f07ad719c36dfd8b5

Note that the only changes we did to the package were to remove the `Process_Element` type and replace the type of the `Proc` parameter of the `Iterate` procedure from a named type (`Process_Element`) to an anonymous type (**not null access procedure** (`E : Element`)).

Now, the same test application we used before (`Show_Access_To_Subprograms`) compiles as expected:

Listing 1570: float_data_processing.ads

```
1 with Data_Processing;
2
3 package Float_Data_Processing is
4     new Data_Processing (Element => Float);
```

Listing 1571: show_access_to_subprograms.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Data_Processing;
4 use Float_Data_Processing;
5
6 procedure Show_Access_To_Subprograms is
7
8     procedure Display (F : Float) is
9     begin
10         Put_Line ("F :" & Float'Image (F));
11     end Display;
12
13     D : Data_Container (5);
14 begin
15     Append (D, 1.0);
16     Append (D, 2.0);
17     Append (D, 3.0);
18
19     Iterate (D, Display'Access);
20 end Show_Access_To_Subprograms;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Anonymous_Access_Types.
↳ Accessibility_Rules_Anonymous_Access_To_Subprograms.Iterator_Anonymous
MD5: 64ee435aac5f2817b7d9cecf538a1e4c

Runtime output

```
F : 1.00000E+00
F : 2.00000E+00
F : 3.00000E+00
```

Remember that the compiler introduces an accessibility check in the call to `Iterate`, which is successful because the lifetime of `Display'Access` is the same as the lifetime of the `Proc` parameter of `Iterate`.

28.3 Limited Types

So far, we discussed nonlimited types in most cases. In this chapter, we discuss limited types.

We can think of limited types as an easy way to avoid inappropriate semantics. For example, a lock should not be copied — neither directly, via assignment, nor with pass-by-copy. Similarly, a *file*, which is really a file descriptor, should not be copied. In this chapter, we'll see example of unwanted side-effects that arise if we don't use limited types for these cases.

28.3.1 Assignment and equality

Limited types have the following restrictions, which we discussed in the *Introduction to Ada* (page 118) course:

- copying objects of limited types via direct assignments is forbidden; and
- there's no predefined equality operator for limited types.

(Of course, in the case of nonlimited types, assignments are possible and the equality operator is available.)

By having these restrictions for limited types, we avoid inappropriate side-effects for assignment and equality operations. As an example of inappropriate side-effects, consider the case when we apply those operations on record types that have components of access types:

Listing 1572: `nonlimited_types.ads`

```

1 package Nonlimited_Types is
2
3   type Simple_Rec is private;
4
5   type Integer_Access is access Integer;
6
7   function Init (I : Integer) return Simple_Rec;
8
9   procedure Set (E : Simple_Rec;
10                I : Integer);
11
12   procedure Show (E      : Simple_Rec;
13                  E_Name : String);
14
15 private
16
17   type Simple_Rec is record
18     V : Integer_Access;
19   end record;
20
21 end Nonlimited_Types;
```

Listing 1573: `nonlimited_types.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 package body Nonlimited_Types is
4
5     function Init (I : Integer) return Simple_Rec
6     is
7     begin
8         return E : Simple_Rec do
9             E.V := new Integer'(I);
10        end return;
11    end Init;
12
13    procedure Set (E : Simple_Rec;
14                  I : Integer) is
15    begin
16        E.V.all := I;
17    end Set;
18
19    procedure Show (E : Simple_Rec;
20                   E_Name : String) is
21    begin
22        Put_Line (E_Name
23                  & ".V.all = "
24                  & Integer'Image (E.V.all));
25    end Show;
26
27 end Nonlimited_Types;

```

Listing 1574: show_wrong_assignment_equality.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Nonlimited_Types; use Nonlimited_Types;
3
4 procedure Show_Wrong_Assignment_Equality is
5     A, B : Simple_Rec := Init (0);
6
7     procedure Show_Compare is
8     begin
9         if A = B then
10            Put_Line ("A = B");
11        else
12            Put_Line ("A /= B");
13        end if;
14    end Show_Compare;
15 begin
16
17     Put_Line ("A := Init (0); A := Init (0);");
18     Show (A, "A");
19     Show (B, "B");
20     Show_Compare;
21     Put_Line ("-----");
22
23     Put_Line ("Set (A, 2); Set (B, 3);");
24     Set (A, 2);
25     Set (B, 3);
26
27     Show (A, "A");
28     Show (B, "B");
29     Put_Line ("-----");
30
31     Put_Line ("B := A");
32     B := A;
33

```

(continues on next page)

(continued from previous page)

```

34 Show (A, "A");
35 Show (B, "B");
36 Show_Compare;
37 Put_Line ("-----");
38
39 Put_Line ("Set (B, 7);");
40 Set (B, 7);
41
42 Show (A, "A");
43 Show (B, "B");
44 Show_Compare;
45 Put_Line ("-----");
46
47 end Show_Wrong_Assignment_Equality;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_Equality.Wrong_Assignment_Equality
 MD5: 72cf7145cd26a8628580c5a837d9cb61

Runtime output

```

A := Init (0); A := Init (0);
A.V.all = 0
B.V.all = 0
A /= B
-----
Set (A, 2); Set (B, 3);
A.V.all = 2
B.V.all = 3
-----
B := A
A.V.all = 2
B.V.all = 2
A = B
-----
Set (B, 7);
A.V.all = 7
B.V.all = 7
A = B
-----
```

In this code, we declare the `Simple_Rec` type in the `Nonlimited_Types` package and use it in the `Show_Wrong_Assignment_Equality` procedure. In principle, we're already doing many things right here. For example, we're declaring the `Simple_Rec` type private, so that the component `V` of access type is encapsulated. Programmers that declare objects of this type cannot simply mess up with the `V` component. Instead, they have to call the `Init` function and the `Set` procedure to initialize and change, respectively, objects of the `Simple_Rec` type. That being said, there are two problems with this code, which we discuss next.

The first problem we can identify is that the first call to `Show_Compare` shows that `A` and `B` are different, although both have the same value in the `V` component (`A.V.all = 0` and `B.V.all = 0`) — this was set by the call to the `Init` function. What's happening here is that the `A = B` expression is comparing the access values (`A.V = B.V`), while we might have been expecting it to compare the actual integer values after dereferencing (`A.V.all = B.V.all`). Therefore, the predefined equality function of the `Simple_Rec` type is useless and dangerous for us, as it misleads us to expect something that it doesn't do.

After the assignment of `A` to `B` (`B := A`), the information that the application displays seems

to be correct — both `A.V.all` and `B.V.all` have the same value of two. However, when assigning the value seven to `B` by calling `Set (B, 7)`, we see that the value of `A.V.all` has also changed. What's happening here is that the previous assignment (`B := A`) has actually assigned access values (`B.V := A.V`), while we might have been expecting it to assign the dereferenced values (`B.V.all := A.V.all`). Therefore, we cannot simply directly assign objects of `Simple_Rec` type, as this operation changes the internal structure of the type due to the presence of components of access type.

For these reasons, forbidding these operations for the `Simple_Rec` type is the most appropriate software design decision. If we still need assignment and equality operators, we can implement custom subprograms for the limited type. We'll discuss this topic in the next sections.

In addition to the case when we have components of access types, limited types are useful for example when we want to avoid the situation in which the same information is copied to multiple objects of the same type.

In the Ada Reference Manual

- 7.5 Limited Types²⁹¹

Assignments

Assignments are forbidden when using objects of limited types. For example:

Listing 1575: `limited_types.ads`

```
1 package Limited_Types is
2
3     type Simple_Rec is limited private;
4
5     type Integer_Access is access Integer;
6
7     function Init (I : Integer) return Simple_Rec;
8
9 private
10
11     type Simple_Rec is limited record
12         V : Integer_Access;
13     end record;
14
15 end Limited_Types;
```

Listing 1576: `limited_types.adb`

```
1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11 end Limited_Types;
```

²⁹¹ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

Listing 1577: show_limited_assignment.adb

```

1 with Limited_Types; use Limited_Types;
2
3 procedure Show_Limited_Assignment is
4   A, B : Simple_Rec := Init (0);
5 begin
6   B := A;
7 end Show_Limited_Assignment;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↳ Equality.Assignment
MD5: 019c16f7feac896fd8c37d40d0522dc8

Build output

```

show_limited_assignment.adb:6:04: error: left hand of assignment must not be_
↳ limited type
gprbuild: *** compilation phase failed

```

In this example, we declare the limited private type `Simple_Rec` and two objects of this type (A and B) in the `Show_Limited_Assignment` procedure. (We discuss more about limited private types *later* (page 1071)).

As expected, we get a compilation error for the `B := A` statement (in the `Show_Limited_Assignment` procedure). If we need to copy two objects of limited type, we have to provide a custom procedure to do that. For example, we can implement a `Copy` procedure for the `Simple_Rec` type:

Listing 1578: limited_types.ads

```

1 package Limited_Types is
2
3   type Integer_Access is access Integer;
4
5   type Simple_Rec is limited private;
6
7   function Init (I : Integer) return Simple_Rec;
8
9   procedure Copy (From : Simple_Rec;
10                  To   : in out Simple_Rec);
11
12 private
13
14   type Simple_Rec is limited record
15     V : Integer_Access;
16   end record;
17
18 end Limited_Types;

```

Listing 1579: limited_types.adb

```

1 package body Limited_Types is
2
3   function Init (I : Integer) return Simple_Rec
4   is
5   begin
6     return E : Simple_Rec do
7       E.V := new Integer'(I);

```

(continues on next page)

(continued from previous page)

```

8     end return;
9 end Init;

10
11 procedure Copy (From : Simple_Rec;
12               To   : in out Simple_Rec)
13 is
14 begin
15     -- Copying record components
16     To.V.all := From.V.all;
17 end Copy;
18
19 end Limited_Types;

```

Listing 1580: show_limited_assignment.adb

```

1 with Limited_Types; use Limited_Types;
2
3 procedure Show_Limited_Assignment is
4     A, B : Simple_Rec := Init (0);
5 begin
6     Copy (From => A, To => B);
7 end Show_Limited_Assignment;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↳ Equality.Assignment
MD5: 2c017c3592c93be8c19fe247e9241fcb

The Copy procedure from this example copies the dereferenced values of From to To, which matches our expectation for the Simple_Rec. Note that we could have also implemented a Shallow_Copy procedure to copy the actual access values (i.e. To.V := From.V). However, having this kind of procedure can be dangerous in many case, so this design decision must be made carefully. In any case, using limited types ensures that only the assignment subprograms that are explicitly declared in the package specification are available.

Equality

Limited types don't have a predefined equality operator. For example:

Listing 1581: limited_types.ads

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;
6
7     function Init (I : Integer) return Simple_Rec;
8
9 private
10
11     type Simple_Rec is limited record
12         V : Integer_Access;
13     end record;
14
15 end Limited_Types;

```

Listing 1582: limited_types.adb

```

1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11 end Limited_Types;

```

Listing 1583: show_limited_equality.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Limited_Types; use Limited_Types;
3
4 procedure Show_Limited_Equality is
5     A : Simple_Rec := Init (5);
6     B : Simple_Rec := Init (6);
7 begin
8     if A = B then
9         Put_Line ("A = B");
10    else
11        Put_Line ("A /= B");
12    end if;
13 end Show_Limited_Equality;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
↳ Equality.Equality
MD5: dad31b5e36de0b3b7824f723a60e5aa0

Build output

```

show_limited_equality.adb:8:09: error: there is no applicable operator "=" for
↳ private type "Simple_Rec" defined at limited_types.ads:5
gprbuild: *** compilation phase failed

```

As expected, the comparison `A = B` triggers a compilation error because no predefined `=` operator is available for the `Simple_Rec` type. If we want to be able to compare objects of this type, we have to implement the `=` operator ourselves. For example, we can do that for the `Simple_Rec` type:

Listing 1584: limited_types.ads

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;
6
7     function Init (I : Integer) return Simple_Rec;
8
9     function "=" (Left, Right : Simple_Rec)
10         return Boolean;
11
12 private

```

(continues on next page)

(continued from previous page)

```

13
14     type Simple_Rec is limited record
15         V : Integer_Access;
16     end record;
17
18 end Limited_Types;

```

Listing 1585: limited_types.adb

```

1 package body Limited_Types is
2
3     function Init (I : Integer) return Simple_Rec
4     is
5     begin
6         return E : Simple_Rec do
7             E.V := new Integer'(I);
8         end return;
9     end Init;
10
11     function "=" (Left, Right : Simple_Rec)
12     return Boolean is
13     begin
14         -- Comparing record components
15         return Left.V.all = Right.V.all;
16     end "=";
17
18 end Limited_Types;

```

Listing 1586: show_limited_equality.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Limited_Types; use Limited_Types;
3
4 procedure Show_Limited_Equality is
5     A : Simple_Rec := Init (5);
6     B : Simple_Rec := Init (6);
7 begin
8     if A = B then
9         Put_Line ("A = B");
10    else
11        Put_Line ("A /= B");
12    end if;
13 end Show_Limited_Equality;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Assignment_
 ↗Equality.Equality
 MD5: f56b2229443a5e4e33c402b41b02d318

Runtime output

```
A /= B
```

Here, the = operator compares the dereferenced values of Left.V and Right.V, which matches our expectation for the Simple_Rec type. Declaring types as limited ensures that we don't have unreasonable equality comparisons, and allows us to create reasonable replacements when required.

i In other languages

In C++, you can overload the assignment operator. For example:

```
class Simple_Rec
{
public:
    // Overloaded assignment
    Simple_Rec& operator= (const Simple_Rec& obj);
private:
    int *V;
};
```

In Ada, however, we can only define the equality operator (=). Defining the assignment operator (:=) is not possible. The following code triggers a compilation error as expected:

```
package Limited_Types is

    type Integer_Access is access Integer;

    type Simple_Rec is limited private;

    procedure "[:=" (To : in out Simple_Rec
                    From : Simple_Rec);

    -- ...

end Limited_Types;
```

28.3.2 Limited private types

As we've seen in code examples from the previous section, we can apply *information hiding* (page 312) to limited types. In other words, we can declare a type as **limited private** instead of just **limited**. For example:

Listing 1587: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is limited record
8         I : Integer;
9     end record;
10
11 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_Types.Limited_Private
 MD5: ececb364f5365a74db43952e9421dee0

In this case, in addition to the fact that assignments are forbidden for objects of this type (because Rec is limited), we cannot access the record components.

Note that in this example, both partial and full views of the Rec record are of limited type. In the next sections, we discuss how the partial and full views can have non-matching declarations.

In the Ada Reference Manual

- 7.5 Limited Types²⁹²

Non-Record Limited Types

In principle, only record types can be declared limited, so we cannot use scalar or array types. For example, the following declarations won't compile:

Listing 1588: non_record_limited_error.ads

```
1 package Non_Record_Limited_Error is
2
3     type Limited_Enumeration is
4         limited (Off, On);
5
6     type Limited_Integer is new
7         limited Integer;
8
9     type Integer_Array is
10         array (Positive range <>) of Integer;
11
12     type Rec is new
13         limited Integer_Array (1 .. 2);
14
15 end Non_Record_Limited_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Non_Record_Limited_Error
MD5: c155e02d809caf28352cbbb579deb861

However, we've mentioned *in a previous chapter* (page 315) that private types don't have to be record types necessarily. In this sense, limited private types makes it possible for us to use types other than record types in the full view and still benefit from the restrictions of limited types. For example:

Listing 1589: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Limited_Enumeration is
4         limited private;
5
6     type Limited_Integer is
7         limited private;
8
9     type Limited_Integer_Array_2 is
10         limited private;
11
12 private
13
14     type Limited_Enumeration is (Off, On);
15
16     type Limited_Integer is new Integer;
17
18     type Integer_Array is
```

(continues on next page)

²⁹² <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

(continued from previous page)

```

19     array (Positive range <>) of Integer;
20
21     type Limited_Integer_Array_2 is
22         new Integer_Array (1 .. 2);
23
24 end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
 ↳Types.Non_Record_Limited
 MD5: 9e65b56a5cb3d7a3da11c7f63ee9bb19

Here, Limited_Enumeration, Limited_Integer, and Limited_Integer_Array_2 are limited private types that encapsulate an enumeration type, an integer type, and a constrained array type, respectively.

Partial and full view of limited types

In the previous example, both partial and full views of the Rec type were limited. We may actually declare a type as **limited private** (in the public part of a package), while its full view is nonlimited. For example:

Listing 1590: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Rec is limited private;
4     -- Partial view of Rec is limited
5
6 private
7
8     type Rec is record
9     -- Full view of Rec is nonlimited
10        I : Integer;
11    end record;
12
13 end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
 ↳Types.Limited_Partial_Full_View
 MD5: 5d0dbc3e87531476856f0ac1f9b22c78

In this case, only the partial view of Rec is limited, while its full view is nonlimited. When deriving from Rec, the view of the derived type is the same as for the parent type:

Listing 1591: simple_recs-child.ads

```

1 package Simple_Recs.Child
2 is
3     type Rec_Derived is new Rec;
4     -- As for its parent, the
5     -- partial view of Rec_Derived
6     -- is limited, but the full view
7     -- is nonlimited.
8
9 end Simple_Recs.Child;

```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Limited_Partial_Full_View
MD5: fdf0ffa87ac2b8766830bf8e17ac7b5e
```

Clients must nevertheless comply with their partial view, and treat the type as if it is in fact limited. In other words, if you use the `Rec` type in a subprogram or package outside of the `Simple_Recs` package (or its child packages), the type is limited from that perspective:

Listing 1592: `use_rec_in_subprogram.adb`

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Use_Rec_In_Subprogram is
4     R1, R2 : Rec;
5 begin
6     R1.I := 1;
7     R2 := R1;
8 end Use_Rec_In_Subprogram;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Limited_Partial_Full_View
MD5: f0af323a951853b97a2b67ce9b13e732
```

Build output

```
use_rec_in_subprogram.adb:6:04: error: invalid prefix in selected component "R1"
use_rec_in_subprogram.adb:7:04: error: left hand of assignment must not be limited_
↳type
gprbuild: *** compilation phase failed
```

Here, compilation fails because the type `Rec` is limited from the procedure's perspective.

Limitations

Note that the opposite — declaring a type as **private** and its full full view as **limited private** — is not possible. For example:

Listing 1593: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Rec is private;
4
5 private
6
7     type Rec is limited record
8         I : Integer;
9     end record;
10
11 end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Limited_Partial_Full_View
MD5: ec1c8a2dcf3cc2c49b1497cf4c9d3a5a
```

Build output

```
simple_recs.ads:7:09: error: completion of nonlimited type cannot be limited
gprbuild: *** compilation phase failed
```

As expected, we get a compilation error in this case. The issue is that the partial view cannot be allowed to mislead the client about what's possible. In this case, if the partial view allows assignment, then the full view must actually provide assignment. But the partial view can restrict what is actually possible, so a limited partial view need not be completed in the full view as a limited type.

In addition, tagged limited private types cannot have a nonlimited full view. For example:

Listing 1594: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec is tagged limited private;
4
5 private
6
7     type Rec is tagged record
8         I : Integer;
9     end record;
10
11 end Simple_Recs;
```

Code block metadata

```
Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Limited_Partial_Full_View
MD5: cadb9ca1346a98fb65f9059fdb29f865
```

Build output

```
simple_recs-child.ads:3:28: error: type derived from tagged type must have
↳extension
simple_recs.ads:7:09: error: completion of limited tagged type must be limited
gprbuild: *** compilation phase failed
```

Here, compilation fails because the type `Rec` is nonlimited in its full view.

Limited and nonlimited in full view

Declaring the full view of a type as limited or nonlimited has implications in the way we can use objects of this type in the package body. For example:

Listing 1595: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Rec_Limited_Full is limited private;
4     type Rec_Nonlimited_Full is limited private;
5
6     procedure Copy
7         (From : Rec_Limited_Full;
8          To   : in out Rec_Limited_Full);
9     procedure Copy
10        (From : Rec_Nonlimited_Full;
11         To   : in out Rec_Nonlimited_Full);
12
13 private
14
```

(continues on next page)

(continued from previous page)

```

15  type Rec_Limited_Full is limited record
16      I : Integer;
17  end record;
18
19  type Rec_Nonlimited_Full is record
20      I : Integer;
21  end record;
22
23  end Simple_Recs;

```

Listing 1596: simple_recs.adb

```

1  package body Simple_Recs is
2
3      procedure Copy
4          (From :      Rec_Limited_Full;
5           To   : in out Rec_Limited_Full)
6      is
7      begin
8          To := From;
9          -- ERROR: assignment is forbidden because
10         --      Rec_Limited_Full is limited in
11         --      its full view.
12      end Copy;
13
14      procedure Copy
15          (From :      Rec_Nonlimited_Full;
16           To   : in out Rec_Nonlimited_Full)
17      is
18      begin
19          To := From;
20          -- OK: assignment is allowed because
21          --      Rec_Nonlimited_Full is
22          --      nonlimited in its full view.
23      end Copy;
24
25  end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_Types.Limited_Non_Limited_Partial_Full_View
 MD5: 24b75bb97ddd485bd6825bb8647607c1

Build output

```

simple_recs.adb:8:07: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

```

Here, both `Rec_Limited_Full` and `Rec_Nonlimited_Full` are declared as **private limited**. However, `Rec_Limited_Full` type is limited in its full view, while `Rec_Nonlimited_Full` is nonlimited. As expected, the compiler complains about the `To := From` assignment in the `Copy` procedure for the `Rec_Limited_Full` type because its full view is limited (so no assignment is possible). Of course, in the case of the objects of `Rec_Nonlimited_Full` type, this assignment is perfectly fine.

Limited private component

Another example mentioned by the Ada Reference Manual (7.3.1²⁹³, 5/1) is about an array type whose component type is limited private, but nonlimited in its full view. Let's see a complete code example for that:

Listing 1597: limited_nonlimited_arrays.ads

```

1 package Limited_Nonlimited_Arrays is
2
3   type Limited_Private is
4     limited private;
5
6   function Init return Limited_Private;
7
8   -- The array type Limited_Private_Array
9   -- is limited because the type of its
10  -- component is limited.
11  type Limited_Private_Array is
12    array (Positive range <>) of
13      Limited_Private;
14
15 private
16
17   type Limited_Private is
18     record
19       A : Integer;
20     end record;
21
22   -- Limited_Private_Array type is
23   -- nonlimited at this point because
24   -- its component is nonlimited.
25   --
26   -- The assignments below are OK:
27   A1 : Limited_Private_Array (1 .. 5);
28
29   A2 : Limited_Private_Array := A1;
30
31 end Limited_Nonlimited_Arrays;
```

Listing 1598: limited_nonlimited_arrays.adb

```

1 package body Limited_Nonlimited_Arrays is
2
3   function Init return Limited_Private is
4     ((A => 1));
5
6 end Limited_Nonlimited_Arrays;
```

Listing 1599: show_limited_nonlimited_array.adb

```

1 with Limited_Nonlimited_Arrays;
2 use Limited_Nonlimited_Arrays;
3
4 procedure Show_Limited_Nonlimited_Array is
5   A3 : Limited_Private_Array (1 .. 2) :=
6     (others => Init);
7   A4 : Limited_Private_Array (1 .. 2);
8 begin
9   -- ERROR: this assignment is illegal because
```

(continues on next page)

²⁹³ <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

(continued from previous page)

```
10  -- Limited_Private_Array is limited, as
11  -- its component is limited at this point.
12  A4 := A3;
13  end Show_Limited_Nonlimited_Array;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Limited_Nonlimited_Array
MD5: 211670e99e6e3a63a785bb2dde255b58

Build output

```
show_limited_nonlimited_array.adb:12:04: error: left hand of assignment must not_
↳be limited type
show_limited_nonlimited_array.adb:12:04: error: component type "Limited_Private"
↳of subtype of "Limited_Private_Array" is limited
gprbuild: *** compilation phase failed
```

As we can see in this example, the limitedness of the array type `Limited_Private_Array` depends on the limitedness of its component type `Limited_Private`. In the private part of `Limited_Nonlimited_Arrays` package, where `Limited_Private` is nonlimited, the array type `Limited_Private_Array` becomes nonlimited as well. In contrast, in the `Show_Limited_Nonlimited_Array`, the array type is limited because its component is limited in that scope.

In the Ada Reference Manual

- 7.3.1 Private Operations²⁹⁴

Tagged limited private types

For tagged private types, the partial and full views must match: if a tagged type is limited in the partial view, it must be limited in the full view. For example:

Listing 1600: `simple_recs.ads`

```
1  package Simple_Recs is
2
3      type Rec is tagged limited private;
4
5  private
6
7      type Rec is tagged limited record
8          I : Integer;
9      end record;
10
11  end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Private_
↳Types.Tagged_Limited_Private_Types
MD5: bee48bd7e0d70ddfd288c0de5e21b039

Here, the tagged `Rec` type is limited both in its partial and full views. Any mismatch in one of the views triggers a compilation error. (As an exercise, you may remove any of the

²⁹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

limited keywords from the code example and try to compile it.)

For further reading...

This rule is for the sake of dynamic dispatching and classwide types. The compiler must not allow any of the types in a derivation class — the set of types related by inheritance — to be different regarding assignment and equality (and thus inequality). That's necessary because we are meant to be able to manipulate objects of any type in the entire set of types via the partial view presented by the root type, without knowing which specific tagged type is involved.

28.3.3 Explicitly limited types

Under certain conditions, limited types can be called explicitly limited — note that using the **limited** keyword in a part of the declaration doesn't necessarily ensure this, as we'll see later.

Let's start with an example of an explicitly limited type:

Listing 1601: simple_rec.s.ads

```

1 package Simple_Recs is
2
3   type Rec is limited record
4     I : Integer;
5   end record;
6
7 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_Types.Explicitly_Limited_Types
 MD5: de73a20140628420830ed9fe0b2dedb5

The Rec type is also explicitly limited when it's declared limited in the private type's completion (in the package's private part):

Listing 1602: simple_rec.s.ads

```

1 package Simple_Recs is
2
3   type Rec is limited private;
4
5 private
6
7   type Rec is limited record
8     I : Integer;
9   end record;
10
11 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_Types.Explicitly_Limited_Types
 MD5: ececb364f5365a74db43952e9421dee0

In this case, Rec is limited both in the partial and in the full view, so it's considered explicitly limited.

However, *as we've learned before* (page 1073), we may actually declare a type as **limited private** in the public part of a package, while its full view is nonlimited. In this case, the limited type is not considered explicitly limited anymore.

For example, if we make the full view of the `Rec` nonlimited (by removing the **limited** keyword in the private part), then the `Rec` type isn't explicitly limited anymore:

Listing 1603: `simple_recs.ads`

```

1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is record
8         I : Integer;
9     end record;
10
11 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Explicitly_Limited_Types
 ↳Types.Explicitly_Limited_Types
 MD5: bd54dec4f9b67d3d14d80511b3ac311f

Now, even though the `Rec` type was declared as limited private, the full view indicates that it's actually a nonlimited type, so it isn't explicitly limited.

Note that *tagged limited private types* (page 1078) are always explicitly limited types — because, as we've learned before, they cannot have a nonlimited type declaration in its full view.

In the Ada Reference Manual

- 6.2 Formal Parameter Modes²⁹⁵
- 6.4.1 Parameter Associations²⁹⁶
- 7.5 Limited Types²⁹⁷

28.3.4 Subtypes of Limited Types

We can declare subtypes of limited types. For example:

Listing 1604: `simple_recs.ads`

```

1 package Simple_Recs is
2
3     type Limited_Integer_Array (L : Positive) is
4         limited private;
5
6     subtype Limited_Integer_Array_2 is
7         Limited_Integer_Array (2);
8
9 private
```

(continues on next page)

²⁹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

²⁹⁶ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

²⁹⁷ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

(continued from previous page)

```

10
11 type Integer_Array is
12     array (Positive range <>) of Integer;
13
14 type Limited_Integer_Array (L : Positive) is
15     limited record
16         Arr : Integer_Array (1 .. L);
17     end record;
18
19 end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Limited_Subtype
MD5: 2a82c3c96fad2a01b9a8c15912d4b974

Here, `Limited_Integer_Array_2` is a subtype of the `Limited_Integer_Array` type. Since `Limited_Integer_Array` is a limited type, the `Limited_Integer_Array_2` subtype is limited as well. A subtype just introduces a name for some constraints on an existing type. As such, a subtype doesn't change the limitedness of the constrained type.

We can test this in a small application:

Listing 1605: test_limitedness.adb

```

1 with Simple_Recs; use Simple_Recs;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Limited_Integer_Array_2;
5 begin
6     Dummy_2 := Dummy_1;
7 end Test_Limitedness;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Limited_Subtype
MD5: c24d07be96f27298a97e18d955cc6161

Build output

```

test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed

```

As expected, compilation fails because `Limited_Integer_Array_2` is a limited (sub)type.

28.3.5 Deriving from limited types

In this section, we discuss the implications of deriving from limited types. As usual, let's start with a simple example:

Listing 1606: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Rec is limited null record;
4
5     type Rec_Derived is new Rec;
6
7 end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Type
MD5: cd23dfb69645ba5f1ebfdd65ee761ebe

In this example, the `Rec_Derived` type is derived from the `Rec` type. Note that the `Rec_Derived` type is limited because its ancestor is limited, even though the **limited** keyword doesn't show up in the declaration of the `Rec_Derived` type. Note that we could have actually used the **limited** keyword here:

```
type Rec_Derived is limited new Rec;
```

Therefore, we cannot use the assignment operator for objects of `Rec_Derived` type:

Listing 1607: test_limitedness.adb

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Test_Limitedness is
4   Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6   Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Type
MD5: ce1b5fc8c96c4ede0cc6768b84296b51

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

Note that we cannot derive a limited type from a nonlimited ancestor:

Listing 1608: simple_recs.ads

```
1 package Simple_Recs is
2
3   type Rec is null record;
4
5   type Rec_Derived is limited new Rec;
6
7 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Type_Nonlimited_Ancestor
MD5: 78a7574cc6233ddc826359acb6e644ee

Build output

```
simple_recs.ads:5:04: error: parent type "Rec" of limited type must be limited
gprbuild: *** compilation phase failed
```

As expected, the compiler indicates that the ancestor `Rec` should be of limited type.

In fact, all types in a derivation class are the same — either limited or not. (That is especially important with dynamic dispatching via tagged types. We discuss this topic in another

chapter.)

i In the Ada Reference Manual

- 7.3 Private Types and Private Extensions²⁹⁸
- 7.5 Limited Types²⁹⁹

Deriving from limited private types

Of course, we can also derive from limited private types. However, there are more rules in this case than the ones we've seen so far. Let's start with an example:

Listing 1609: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is limited null record;
8
9 end Simple_Recs;
```

Listing 1610: simple_recs-ext.ads

```

1 package Simple_Recs.Ext is
2
3     type Rec_Derived is new Rec;
4
5     -- OR:
6     --
7     -- type Rec_Derived is
8     --     limited new Rec;
9
10 end Simple_Recs.Ext;
```

Listing 1611: test_limitedness.adb

```

1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Private_Type
MD5: c6eed14520589b9c1e11c17bd6179c19

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

²⁹⁸ <http://www.ada-auth.org/standards/22rm/html/RM-7-3.html>

²⁹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

Here, `Rec_Derived` is a limited type derived from the (limited private) `Rec` type. We can verify that `Rec_Derived` type is limited because the compilation of the `Test_Limitedness` procedure fails.

Deriving from non-explicitly limited private types

Up to this point, we have discussed *explicitly limited types* (page 1079). Now, let's see how derivation works with *non-explicitly* limited types.

Any type derived from a limited type is always limited, even if the full view of its ancestor is nonlimited. For example, let's modify the full view of `Rec` and make it nonlimited (i.e. make it *not explicitly* limited):

Listing 1612: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Rec is limited private;
4
5 private
6
7     type Rec is null record;
8
9 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Private_Type
MD5: 30a2a88ff46b7e528bb8d75d3d6ad6ce

Build output

```
simple_recs.ads:1: Simple_Recs cannot be used as a main program
gprbind: invocation of gnatbind failed
gprbuild: unable to bind simple_recs.ads
```

Here, `Rec_Derived` is a limited type because the partial view of `Rec` is limited. The fact that the full view of `Rec` is nonlimited doesn't affect the `Rec_Derived` type — as we can verify with the compilation error in the `Test_Limitedness` procedure.

Note, however, that a derived type becomes nonlimited in the **private part or the body** of a child package if it isn't explicitly limited. In this sense, the derived type inherits the *nonlimitedness* of the parent's full view. For example, because we're declaring `Rec_Derived` as **is new** `Rec` in the child package (`Simple_Recs.Ext`), we're saying that `Rec_Derived` is limited *outside* this package, but nonlimited in the private part and body of the `Simple_Recs.Ext` package. We can verify this by copying the code from the `Test_Limitedness` procedure to a new procedure in the body of the `Simple_Recs.Ext` package:

Listing 1613: `simple_recs-ext.ads`

```
1 package Simple_Recs.Ext
2     with Elaborate_Body is
3
4     -- Rec_Derived is derived from Rec, which is a
5     -- limited private type that is nonlimited in
6     -- its full view.
7     --
8     -- Rec_Derived isn't explicitly limited.
9     -- Therefore, it's nonlimited in the private
10    -- part of Simple_Recs.Ext and its package
```

(continues on next page)

(continued from previous page)

```

11  -- body.
12  --
13  type Rec_Derived is new Rec;
14
15  end Simple_Recs.Ext;

```

Listing 1614: simple_recs-ext.adb

```

1  package body Simple_Recs.Ext is
2
3      procedure Test_Child_Limitedness is
4          Dummy_1, Dummy_2 : Rec_Derived;
5      begin
6          -- Here, Rec_Derived is a nonlimited
7          -- type because Rec is nonlimited in
8          -- its full view.
9
10         Dummy_2 := Dummy_1;
11     end Test_Child_Limitedness;
12
13 end Simple_Recs.Ext;

```

Listing 1615: test_limitedness.adb

```

1  -- We copied the code to the
2  -- Test_Child_Limitedness procedure (in the
3  -- body of the Simple_Recs.Ext package) and
4  -- commented it out here.
5  --
6  -- You may uncomment the code to verify
7  -- that Rec_Derived is limited in this
8  -- procedure.
9  --
10
11 -- with Simple_Recs.Ext; use Simple_Recs.Ext;
12
13 procedure Test_Limitedness is
14     -- Dummy_1, Dummy_2 : Rec_Derived;
15 begin
16     -- Dummy_2 := Dummy_1;
17     null;
18 end Test_Limitedness;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Limited_Private_Type
MD5: f480cd05afff622e451684a0293cb982

```

In the `Test_Child_Limitedness` procedure of the `Simple_Recs.Ext` package, we can use the `Rec_Derived` as a nonlimited type because its ancestor `Rec` is nonlimited in its full view. (*As we've learned before* (page 1075), if a limited type is nonlimited in its full view, we can copy objects of this type in the private part of the package specification or in the package body.)

Outside of the package, both `Rec` and `Rec_Derived` types are limited types. Therefore, if we uncomment the code in the `Test_Limitedness` procedure, compilation fails there (because `Rec_Derived` is viewed as descending from a limited type).

Deriving from tagged limited private types

The rules for deriving from tagged limited private types are slightly different than the rules we've seen so far. This is because tagged limited types are always *explicitly limited types* (page 1079).

Let's look at an example:

Listing 1616: simple_recs.ads

```
1 package Simple_Recs is
2
3     type Tagged_Rec is tagged limited private;
4
5 private
6
7     type Tagged_Rec is tagged limited null record;
8
9 end Simple_Recs;
```

Listing 1617: simple_recs-ext.ads

```
1 package Simple_Recs.Ext is
2
3     type Rec_Derived is new
4         Tagged_Rec with private;
5
6 private
7
8     type Rec_Derived is new
9         Tagged_Rec with null record;
10
11 end Simple_Recs.Ext;
```

Listing 1618: test_limitedness.adb

```
1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4     Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6     Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Tagged_Limited_Private_Type
MD5: 81c8a010f093d8823b84bb6e69c4114e

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

In this example, `Rec_Derived` is a tagged limited type derived from the `Tagged_Rec` type. (Again, we can verify the limitedness of the `Rec_Derived` type with the `Test_Limitedness` procedure.)

As explained previously, the derived type (`Rec_Derived`) is a limited type, even though the **limited** keyword doesn't appear in its declaration. We could, of course, include the **limited** keyword in the declaration of `Rec_Derived`:

Listing 1619: simple_recs-ext.ads

```

1 package Simple_Recs.Ext is
2
3     type Rec_Derived is limited new
4       Tagged_Rec with private;
5
6 private
7
8     type Rec_Derived is limited new
9       Tagged_Rec with null record;
10
11 end Simple_Recs.Ext;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Tagged_Limited_Private_Type
MD5: b82a58a4bf9701b321000c52bf121977

Build output

```

simple_recs-ext.ads:1: Simple_Recs.ext cannot be used as a main program
gprbind: invocation of gnatbind failed
gprbuild: unable to bind simple_recs-ext.ads
```

(Obviously, if we include the **limited** keyword in the partial view of the derived type, we must include it in its full view as well.)

Deriving from limited interfaces

The rules for limited interfaces are different from the ones for limited tagged types. In contrast to the rule we've seen in the previous section, a type that is derived from a limited type isn't automatically limited. In other words, it does **not** inherit the *limitedness* from the interface. For example:

Listing 1620: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Limited_IF is limited interface;
4
5 end Simple_Recs;
```

Listing 1621: simple_recs-ext.ads

```

1 package Simple_Recs.Ext is
2
3     type Rec_Derived is new
4       Limited_IF with private;
5
6 private
7
8     type Rec_Derived is new
9       Limited_IF with null record;
10
11 end Simple_Recs.Ext;
```

Listing 1622: test_limitedness.adb

```
1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4   Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6   Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Interface_Limited_Private
MD5: d9cf0bd26b86d0caec82eff2a2ec6ead

Here, Rec_Derived is derived from the limited Limited_IF interface. As we can see, the Test_Limitedness compiles fine because Rec_Derived is nonlimited.

Of course, if we want Rec_Derived to be limited, we can make this explicit in the type declaration:

Listing 1623: simple_recs-ext.ads

```
1 package Simple_Recs.Ext is
2
3   type Rec_Derived is limited new
4     Limited_IF with private;
5
6 private
7
8   type Rec_Derived is limited new
9     Limited_IF with null record;
10
11 end Simple_Recs.Ext;
```

Listing 1624: test_limitedness.adb

```
1 with Simple_Recs.Ext; use Simple_Recs.Ext;
2
3 procedure Test_Limitedness is
4   Dummy_1, Dummy_2 : Rec_Derived;
5 begin
6   Dummy_2 := Dummy_1;
7 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Deriving_From_
↳ Limited_Types.Derived_Interface_Limited_Private
MD5: abb295cbfd5ade5f351991c2fbaf519c

Build output

```
test_limitedness.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

Now, compilation of Test_Limitedness fails because Rec_Derived is explicitly limited.

28.3.6 Immutably Limited Types

According to the [Annotated Ada Reference Manual \(7.5, 8.b/3\)](#)³⁰⁰, "an immutably limited type is a type that cannot become nonlimited subsequently in a private part or in a child unit." In fact, while we were talking about *partial and full view of limited types* (page 1073), we've seen that limited private types can become nonlimited in their full view. Such limited types are *not* immutably limited.

The Annotated Ada Reference Manual also says that "if a view of the type makes it immutably limited, then no copying (assignment) operations are ever available for objects of the type. This allows other properties; for instance, it is safe for such objects to have access discriminants that have defaults or designate other limited objects." We'll see examples of this later on.

Immutably limited types include:

- *explicitly limited types* (page 1079)
- tagged limited types (i.e. with the keywords **tagged limited**);
- *tagged limited private types* (page 1078);
- limited private type that have at least one *access discriminant* (page 1011) with a default expression;
- task types, protected types, and synchronized interfaces;
- any types derived from immutably limited types.

Let's look at a code example that shows instances of immutably limited types:

Listing 1625: show_immutably_limited_types.ads

```

1 package Show_Immutably_Limited_Types is
2
3   --
4   -- Explicitly limited type
5   --
6   type Explicitly_Limited_Rec is limited
7   record
8     A : Integer;
9   end record;
10
11  --
12  -- Tagged limited type
13  --
14  type Limited_Tagged_Rec is tagged limited
15  record
16    A : Integer;
17  end record;
18
19  --
20  -- Tagged limited private type
21  --
22  type Limited_Tagged_Private is
23    tagged limited private;
24
25  --
26  -- Limited private type with an access
27  -- discriminant that has a default
28  -- expression
29  --
30  type Limited_Rec_Access_D

```

(continues on next page)

³⁰⁰ <http://www.ada-auth.org/standards/22aarm/html/AA-7-5.html>

(continued from previous page)

```

31     (AI : access Integer := new Integer) is
32         limited private;
33
34     --
35     -- Task type
36     --
37     task type TT is
38         entry Start;
39         entry Stop;
40     end TT;
41
42     --
43     -- Protected type
44     --
45     protected type PT is
46         function Value return Integer;
47     private
48         A : Integer;
49     end PT;
50
51     --
52     -- Synchronized interface
53     --
54     type SI is synchronized interface;
55
56     --
57     -- A type derived from an immutably
58     -- limited type
59     --
60     type Derived_Immutable is new
61         Explicitly_Limited_Rec;
62
63 private
64
65     type Limited_Tagged_Private is tagged limited
66     record
67         A : Integer;
68     end record;
69
70     type Limited_Rec_Access_D
71     (AI : access Integer := new Integer)
72     is limited
73     record
74         A : Integer;
75     end record;
76
77 end Show_Immutable_Limited_Types;

```

Listing 1626: show_immutably_limited_types.adb

```

1 package body Show_Immutable_Limited_Types is
2
3     task body TT is
4     begin
5         accept Start;
6         accept Stop;
7     end TT;
8
9     protected body PT is
10         function Value return Integer is
11             (PT.A);

```

(continues on next page)

(continued from previous page)

```

12   end PT;
13
14 end Show_Immutable_Limited_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Immutable_Limited_Types.Example
 MD5: 6bcb9582a10eedc96040ab11cd320153

Build output

```
show_immutable_limited_types.ads:31:30: warning: coextension will not be
↳ deallocated when its associated owner is deallocated [enabled by default]
```

In the `Show_Immutable_Limited_Types` package above, we see multiple instances of immutably limited types. (The comments in the source code indicate each type.)

In the Ada Reference Manual

- 7.5 Limited Types³⁰¹

Non immutably limited types

Not every limited type is immutably limited. We already mentioned untagged private limited types, which can *become nonlimited in their full view* (page 1073). In addition, we have nonsynchronized limited interface types. As mentioned earlier in this chapter, a *type derived from a nonsynchronized limited interface* (page 1087), can be nonlimited, so it's not immutably limited.

In the Ada Reference Manual

- 7.3.1 Private Operations³⁰²
- 7.5 Limited Types³⁰³

28.3.7 Limited Types with Discriminants

In this section, we look into the implications of using discriminants with limited types. Actually, most of the topics mentioned here have already been covered in different sections of previous chapters, as well as in this chapter. Therefore, this section is in most parts just a review of what we've already discussed.

Let's start with a simple example:

Listing 1627: `simple_recs.ads`

```

1 package Simple_Recs is
2
3   type Rec (L : Positive)
4     is limited null record;
5
6 end Simple_Recs;
```

³⁰¹ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

³⁰² <http://www.ada-auth.org/standards/22rm/html/RM-7-3-1.html>

³⁰³ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

Listing 1628: test_limitedness.adb

```

1 with Simple_Recs; use Simple_Recs;
2
3 procedure Test_Limitedness is
4   Dummy_1 : Rec (2);
5   Dummy_2 : Rec (3);
6 begin
7   Dummy_2 := Dummy_1;
8   -- ^^^^^^^^^^^^^^^
9   -- ERRORS:
10  -- 1. Cannot assign objects of
11  --    limited types.
12  -- 2. Cannot assign objects with
13  --    different discriminants.
14 end Test_Limitedness;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳ Simple_Example
 MD5: 7b4a62c0341becf16f59e163b4359397

Build output

```
test_limitedness.adb:7:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

In this example, we see the declaration of the limited type `Rec`, which has the discriminant `L`. For objects of type `Rec`, we not only have the typical restrictions that *equality and assignment aren't available* (page 1066), but we also have the restriction that we won't be able to assign objects with different discriminants.

** In the Ada Reference Manual**

- [3.7 Discriminants](#)³⁰⁴

Default Expressions

On the other hand, there are restrictions that apply to nonlimited types with discriminants, but not to limited types with discriminants. This concerns mostly default expressions, which are generally allowed for discriminants of limited types.

Discriminants of tagged limited types

As we've discussed previously, we can use default expressions for discriminants of tagged limited types. Let's see an example:

Listing 1629: recs.ads

```

1 package Recs is
2
3   type LTT (L : Positive := 1;
4             M : Positive := 2) is
5     tagged limited null record;
6
7 end Recs;
```

³⁰⁴ <http://www.ada-auth.org/standards/12rm/html/RM-3-7.html>

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳Discriminant_Default_Value_Tagged_Type
 MD5: ebd28ee124c6a84765c61ea609ba0595

Obviously, the same applies to *tagged limited private types* (page 1078):

Listing 1630: recs.ads

```

1 package Recs is
2
3     type LTT (L : Positive := 1;
4               M : Positive := 2) is
5         tagged limited private;
6
7 private
8
9     type LTT (L : Positive := 1;
10              M : Positive := 2) is
11         tagged limited null record;
12
13 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳Discriminant_Default_Value_Tagged_Type
 MD5: dd3d5a25ad9d050f7e7467d859dd9e14

In the case of tagged, nonlimited types, using default expressions in this context isn't allowed.

Access discriminant

Similarly, when using limited types, we can specify default expressions for *access discriminants* (page 1011):

Listing 1631: custom_rec.s.ads

```

1 package Custom_Recs is
2
3     -- Specifying a default expression for
4     -- an access discriminant:
5     type Rec (IA : access Integer :=
6               new Integer'(0)) is limited
7     record
8         I : Integer := IA.all;
9     end record;
10
11 end Custom_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳Access_Discriminant_Default_Expression
 MD5: 23703d9dc80e9f1c8fe237c76b9dd6b0

Build output

```

custom_rec.s.ads:6:21: warning: coextension will not be deallocated when its
↳associated owner is deallocated [enabled by default]
```


In fact, *as we've discussed before* (page 1013), this isn't possible for nonlimited types.

Note, however, that we can only assign a default expression to an access discriminant of an *immutably limited type* (page 1089).

Discriminants of nontagged limited types

In addition to tagged limited types, we can use default expressions for discriminants of nontagged limited types. Let's see an example:

Listing 1632: recs.ads

```
1 package Recs is
2
3     type LTT (L : Positive := 1;
4               M : Positive := 2) is
5         limited null record;
6
7 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↳Discriminant_Default_Value_Tagged_Type
MD5: 8d189b814c6afb4f060e9a41558c18c6

Obviously, the same applies to *limited private types* (page 1071):

Listing 1633: recs.ads

```
1 package Recs is
2
3     type LTT (L : Positive := 1;
4               M : Positive := 2) is
5         limited private;
6
7 private
8
9     type LTT (L : Positive := 1;
10              M : Positive := 2) is
11         limited null record;
12
13 end Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↳Discriminant_Default_Value_Tagged_Type
MD5: 536d7dfbe84c818cb94a8b972e3d77cb

Note that using default expressions for discriminants of nonlimited, nontagged types is OK as well.

Mutable subtypes and Limitedness

As we've mentioned before, an unconstrained discriminated subtype with defaults is called a mutable subtype. An important feature of mutable subtypes is that it allows changing the discriminants of an object, e.g. via assignments. However, as we know, we cannot assign to objects of limited types. Therefore, in essence, a type should be nonlimited to be considered a mutable subtype.

Let's look at a code example:

Listing 1634: recs.ads

```

1 package Recs is
2
3   type LTT (L : Positive := 1;
4             M : Positive := 2) is
5     limited null record;
6
7   function Init (L : Positive;
8                 M : Positive)
9     return LTT is
10    ((L => L, M => M));
11
12   procedure Copy (From : LTT;
13                  To : in out LTT);
14
15 end Recs;

```

Listing 1635: recs.adb

```

1 package body Recs is
2
3   procedure Copy (From : LTT;
4                  To : in out LTT) is
5   begin
6     To := Init (L => From.L,
7                M => From.M);
8     -- ERROR: cannot assign to object of
9     --         limited type
10
11     To.L := From.L;
12     To.M := From.M;
13     -- ERROR: cannot change discriminants
14   end Copy;
15
16 end Recs;

```

Listing 1636: show.adb

```

1 with Recs; use Recs;
2
3 procedure Show is
4   A : LTT;
5   B : LTT := Init (10, 12);
6 begin
7   Copy (From => B, To => A);
8 end Show;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
 ↳Discriminant_Default_Value_Tagged_Type
 MD5: e8dfb1e99e33923aa4023428ecb17372

Build output

```

recs.adb:6:07: error: left hand of assignment must not be limited type
recs.adb:11:09: error: assignment to discriminant not allowed
recs.adb:12:09: error: assignment to discriminant not allowed
gprbuild: *** compilation phase failed

```

As we can see in the Copy procedure, it's not possible to properly assign to the target

object. Using `Init` is forbidden because the assignment is not initializing the target object — as we're not declaring `To` at this point. Also, changing the individual discriminants is forbidden as well. Therefore, we don't have any means to change the discriminants of the target object. (In contrast, if `LTT` was a nonlimited type, we would be able to implement `Copy` by using the call to the `Init` function.)

Limited private type with unknown discriminants

We can declare limited private types with *unknown discriminants* (page 488). Let's see an example:

Listing 1637: `limited_private_unknown_discriminants.ads`

```
1 package Limited_Private_Unknown_Discriminants is
2
3     type Rec (<>) is limited private;
4
5 private
6
7     type Rec is limited
8     record
9         I : Integer;
10    end record;
11
12 end Limited_Private_Unknown_Discriminants;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↳ Limited_Private_Unknown_Discriminants
MD5: 74184919132a084da76bd3e1445c22e5

In this example, we declare type `Rec`, which has unknown discriminants.

As we mentioned earlier, when we use a private type with unknown discriminants, we gain extra control over its initialization. In addition, if we declare those types as limited, we gain even more control. In fact, this is what the *Annotated Ada Reference Manual* (3.7, 26.b/2)³⁰⁵ says:

"A subtype with unknown discriminants is indefinite, and hence an object of such a subtype needs explicit initialization. A limited private type with unknown discriminants is 'extremely' limited; objects of such a type can be initialized only by subprograms (either procedures with a parameter of the type, or a function returning the type) declared in the package. Subprograms declared elsewhere can operate on and even return the type, but they can only initialize the object by calling (ultimately) a subprogram in the package declaring the type. Such a type is useful for keeping complete control over object creation within the package declaring the type."

Let's reuse a code example from the *previous section on unknown discriminants* (page 490) and use limited types:

Listing 1638: `limited_private_unknown_discriminants.ads`

```
1 package Limited_Private_Unknown_Discriminants is
2
3     type Rec (<>) is limited private;
4
5     function Init return Rec;
6
```

(continues on next page)

³⁰⁵ <http://www.ada-auth.org/standards/22aarm/html/AA-3-7.html>

(continued from previous page)

```

7 private
8
9     type Rec is limited
10    record
11        I : Integer;
12    end record;
13
14    function Init return Rec is
15        ((I => 0));
16
17 end Limited_Private_Unknown_Discriminants;

```

Listing 1639: show_constructor_function.adb

```

1 with Limited_Private_Unknown_Discriminants;
2 use Limited_Private_Unknown_Discriminants;
3
4 procedure Show_Constructor_Function is
5     R : Rec := Init;
6 begin
7     null;
8 end Show_Constructor_Function;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Discriminants.
↳ Limited_Private_Unknown_Discriminants
MD5: f4b1de2a83837e2e52b0b57214f0eaf9

A function such as Init is called a *constructor function for limited types* (page 1102). We discuss this topic in more detail later on.

28.3.8 Record components of limited type

In this section, we discuss the implications of using components of limited type. Let's start by declaring a record component of limited type:

Listing 1640: simple_recs.ads

```

1 package Simple_Recs is
2
3     type Int_Rec is limited record
4         V : Integer;
5     end record;
6
7     type Rec is limited record
8         IR : Int_Rec;
9     end record;
10
11 end Simple_Recs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Record_Components_
↳ Limited_Type.Record_Components_Limited_Type
MD5: 71badd1e38cc4ff37f16d99dd203614b

As soon as we declare a record component of some limited type, the whole record is limited. In this example, the Rec record is limited due to the presence of the IR component of limited type.

Also, if we change the declaration of the `Rec` record from the previous example and remove the **limited** keyword, the type itself remains implicitly limited. We can see that when trying to assign to objects of `Rec` type in the `Show_Implicitly_Limited` procedure:

Listing 1641: `simple_recs.ads`

```
1 package Simple_Recs is
2
3     type Int_Rec is limited record
4         V : Integer;
5     end record;
6
7     type Rec is record
8         IR : Int_Rec;
9     end record;
10
11 end Simple_Recs;
```

Listing 1642: `show_implicitly_limited.adb`

```
1 with Simple_Recs; use Simple_Recs;
2
3 procedure Show_Implicitly_Limited is
4     A, B : Rec;
5 begin
6     B := A;
7 end Show_Implicitly_Limited;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Record_Components_
↳ Limited_Type.Record_Components_Limited_Type
MD5: 39770daecfc4579407a799e14f9feff9

Build output

```
show_implicitly_limited.adb:6:04: error: left hand of assignment must not be limited type
↳ limited type
show_implicitly_limited.adb:6:04: error: component "IR" of type "Rec" has limited type
↳ type
gprbuild: *** compilation phase failed
```

Here, the compiler indicates that the assignment is forbidden because the `Rec` type has a component of limited type. The rationale for this rule is that an object of a limited type doesn't allow assignment or equality, including the case in which that object is a component of some enclosing composite object. If we allowed the enclosing object to be copied or tested for equality, we'd be doing it for all the components, too.

In the Ada Reference Manual

- [3.8 Record Types](#)³⁰⁶

28.3.9 Limited types and aggregates

Note

³⁰⁶ <http://www.ada-auth.org/standards/22rm/html/RM-3-8.html>

This section was originally written by Robert A. Duff and published as [Gem #1: Limited Types in Ada 2005](#)³⁰⁷ and [Gem #2](#)³⁰⁸.

In this section, we focus on using aggregates to initialize limited types.

Historically

Prior to Ada 2005, aggregates were illegal for limited types. Therefore, we would be faced with a difficult choice: Make the type limited, and initialize it like this:

Listing 1643: persons.ads

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package Persons is
5
6     type Limited_Person;
7     type Limited_Person_Access is
8       access all Limited_Person;
9
10    type Limited_Person is limited record
11      Name      : Unbounded_String;
12      Age       : Natural;
13    end record;
14
15 end Persons;
```

Listing 1644: show_non_aggregate_init.adb

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 with Persons; use Persons;
5
6 procedure Show_Non_Aggregate_Init is
7   X : Limited_Person;
8 begin
9   X.Name := To_Unbounded_String ("John Doe");
10  X.Age := 25;
11 end Show_Non_Aggregate_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.Full_Coverage_Rules_Limited_Ada95
 MD5: fd3dcb6251f7b6912dafcca052932be2

which has the maintenance problem the full coverage rules are supposed to prevent. Or, make the type nonlimited, and gain the benefits of aggregates, but lose the ability to prevent copies.

Full coverage rules for limited types

Previously, we discussed *full coverage rules for aggregates* (page 529). They also apply to limited types.

³⁰⁷ <https://www.adacore.com/gems/gem-1>

³⁰⁸ <https://www.adacore.com/gems/gem-2>

Historically

The full coverage rules have been aiding maintenance since Ada 83. However, prior to Ada 2005, we couldn't use them for limited types.

Suppose we have the following limited type:

Listing 1645: persons.ads

```
1 with Ada.Strings.Unbounded;  
2 use Ada.Strings.Unbounded;  
3  
4 package Persons is  
5  
6   type Limited_Person;  
7   type Limited_Person_Access is  
8     access all Limited_Person;  
9  
10  type Limited_Person is limited record  
11    Self : Limited_Person_Access :=  
12      Limited_Person'Unchecked_Access;  
13    Name : Unbounded_String;  
14    Age  : Natural;  
15    Shoe_Size : Positive;  
16  end record;  
17  
18 end Persons;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
↳ Aggregates.Full_Coverage_Rules_Limited
MD5: b8ece44a10d512061cb138be21e42034

This type has a self-reference; it doesn't make sense to copy objects, because Self would end up pointing to the wrong place. Therefore, we would like to make the type limited, to prevent developers from accidentally making copies. After all, the type is probably private, so developers using this package might not be aware of the problem. We could also solve that problem with controlled types, but controlled types are expensive, and add unnecessary complexity if not needed.

We can initialize objects of limited type with an aggregate. Here, we can say:

Listing 1646: show_aggregate_box_init.adb

```
1 with Ada.Strings.Unbounded;  
2 use Ada.Strings.Unbounded;  
3  
4 with Persons; use Persons;  
5  
6 procedure Show_Aggregate_Box_Init is  
7   X : aliased Limited_Person :=  
8     (Self      => <>,  
9       Name     =>  
10        To_Unbounded_String ("John Doe"),  
11       Age      => 25,  
12       Shoe_Size => 10);  
13 begin  
14   null;  
15 end Show_Aggregate_Box_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.Full_Coverage_Rules_Limited
 MD5: ded40ff29b53ea5528efba94efaadbec

The `Self => <>` means use the default value of `Limited_Person'Unchecked_Access`. Since `Limited_Person` appears inside the type declaration, it refers to the "current instance" of the type, which in this case is `X`. Thus, we are setting `X.Self` to be `X'Unchecked_Access`.

One very important requirement should be noted: the implementation is required to build the value of `X` *in place*; it cannot construct the aggregate in a temporary variable and then copy it into `X`, because that would violate the whole point of limited objects — you can't copy them.

Historically

Since Ada 2005, an aggregate is allowed to be limited; we can say:

Listing 1647: show_aggregate_init.adb

```

1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3 with Persons; use Persons;
4
5 procedure Show_Aggregate_Init is
6
7     X : aliased Limited_Person :=
8         (Self      => null, -- Wrong!
9           Name      =>
10              To_Unbounded_String ("John Doe"),
11           Age       => 25,
12           Shoe_Size => 10);
13
14 begin
15     X.Self := X'Unchecked_Access;
16 end Show_Aggregate_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.Full_Coverage_Rules_Limited
 MD5: 793ee000fd777d0aa5c15e16132ec411

It seems uncomfortable to set the value of `Self` to the wrong value (`null`) and then correct it. It also seems annoying that we have a (correct) default value for `Self`, but prior to Ada 2005, we couldn't use defaults with aggregates. Since Ada 2005, a new syntax in aggregates is available: `<>` means "use the default value, if any". Therefore, we can replace `Self => null` by `Self => <>`.

Important

Note that using `<>` in an aggregate can be dangerous, because it can leave some components uninitialized. `<>` means "use the default value". If the type of a component is scalar, and there is no record-component default, then there is no default value.

For example, if we have an aggregate of type **String**, like this:

Listing 1648: show_string_box_init.adb

```

1 procedure Show_String_Box_Init is
2   Uninitialized_Const_Str : constant String :=
3     (1 .. 10 => <>);
4 begin
5   null;
6 end Show_String_Box_Init;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.String_Box_Init
 MD5: 28931ced4e1113d55bdc9dc64b42f70a

we end up with a 10-character string all of whose characters are invalid values. Note that this is no more nor less dangerous than this:

Listing 1649: show_dangerous_string.adb

```

1 procedure Show_Dangerous_String is
2   Uninitialized_String_Var : String (1 .. 10);
3   -- ~~~~~
4   -- no initialization
5
6   Uninitialized_Const_Str : constant String :=
7     Uninitialized_String_Var;
8 begin
9   null;
10 end Show_Dangerous_String;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↳Aggregates.Dangerous_String
 MD5: 6c26e9c8d5d031d4e6eac1ac8458f17e

Build output

show_dangerous_string.adb:2:05: warning: variable "Uninitialized_String_Var" is
 ↳read but never assigned [-gnatvw]

As always, one must be careful about uninitialized scalar objects.

28.3.10 Constructor functions for limited types

Note

This section was originally written by Robert A. Duff and published as [Gem #3³⁰⁹](#).

Given that we can use build-in-place aggregates for limited types, the obvious next step is to allow such aggregates to be wrapped in an abstraction — namely, to return them from functions. After all, interesting types are usually private, and we need some way for clients to create and initialize objects.

Historically

Prior to Ada 2005, constructor functions (that is, functions that create new objects and return them) were not allowed for limited types. Since Ada 2005, fully-general construc-

³⁰⁹ <https://www.adacore.com/gems/gem-3>

tor functions are allowed.

Let's see an example:

Listing 1650: p.ads

```

1  with Ada.Strings.Unbounded;
2  use  Ada.Strings.Unbounded;
3
4  package P is
5      task type Some_Task_Type;
6
7      protected type Some_Protected_Type is
8          -- dummy type
9      end Some_Protected_Type;
10
11     type T (<>) is limited private;
12     function Make_T (Name : String) return T;
13     --
14     -- constructor function
15 private
16     type T is limited
17         record
18             Name      : Unbounded_String;
19             My_Task    : Some_Task_Type;
20             My_Prot    : Some_Protected_Type;
21         end record;
22 end P;
```

Listing 1651: p.adb

```

1  package body P is
2
3      task body Some_Task_Type is
4          begin
5              null;
6          end Some_Task_Type;
7
8      protected body Some_Protected_Type is
9          end Some_Protected_Type;
10
11     function Make_T (Name : String) return T is
12     begin
13         return (Name =>
14                 To_Unbounded_String (Name),
15                 others => <>);
16     end Make_T;
17
18 end P;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
 Functions_Limited_Types.Constructor_Functions
 MD5: 2e73eea0ba7852d45ba96dc1f6fae14d

Given the above, clients can say:

Listing 1652: show_constructor_function.adb

```

1 with P; use P;
2
3 procedure Show_Constructor_Function is
4   My_T : T := Make_T
5           (Name => "Bartholomew Cubbins");
6 begin
7   null;
8 end Show_Constructor_Function;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
 ↪ Functions_Limited_Types.Constructor_Functions
 MD5: 52801fafbd58fedbf268a6704008627b

As for aggregates, the result of `Make_T` is built in place (that is, in `My_T`), rather than being created and then copied into `My_T`. Adding another level of function call, we can do:

Listing 1653: show_rumplestiltskin_constructor.adb

```

1 with P; use P;
2
3 procedure Show_Rumplestiltskin_Constructor is
4
5   function Make_Rumplestiltskin return T is
6   begin
7     return Make_T (Name => "Rumplestiltskin");
8   end Make_Rumplestiltskin;
9
10  Rumplestiltskin_Is_My_Name : constant T :=
11    Make_Rumplestiltskin;
12 begin
13   null;
14 end Show_Rumplestiltskin_Constructor;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
 ↪ Functions_Limited_Types.Constructor_Functions
 MD5: d8d9e9f22a0f2f034057fe97f75eacfe

It might help to understand the implementation model: In this case, `Rumplestiltskin_Is_My_Name` is allocated in the usual way (on the stack, presuming it is declared local to some subprogram). Its address is passed as an extra implicit parameter to `Make_Rumplestiltskin`, which then passes that same address on to `Make_T`, which then builds the aggregate in place at that address. Limited objects must never be copied! In this case, `Make_T` will initialize the `Name` component, and create the `My_Task` and `My_Prot` components, all directly in `Rumplestiltskin_Is_My_Name`.

Historically

Note that `Rumplestiltskin_Is_My_Name` is constant. Prior to Ada 2005, it was impossible to create a constant limited object, because there was no way to initialize it.

As we discussed before (page 1096), the ($\leftrightarrow$) on type `T` means that it has *unknown discriminants* from the point of view of the client. This is a trick that prevents clients from creating default-initialized objects (that is, `X : T;` is illegal). Thus clients must call `Make_T`

whenever an object of type T is created, giving package P full control over initialization of objects.

Ideally, limited and nonlimited types should be just the same, except for the essential difference: you can't copy limited objects (and there's no language-defined equality operator). By allowing functions and aggregates for limited types, we're very close to this goal. Some languages have a specific feature called *constructor*. In Ada, a *constructor* is just a function that creates a new object.

Historically

Prior to Ada 2005, *constructors* only worked for nonlimited types. For limited types, the only way to *construct* on declaration was via default values, which limits you to one constructor. And the only way to pass parameters to that construction was via discriminants.

Consider the following package:

Listing 1654: aux.ads

```

1 with Ada.Containers.Ordered_Sets;
2
3 package Aux is
4   generic
5     with package OS is new
6       Ada.Containers.Ordered_Sets (<>);
7   function Gen_Singleton_Set
8     (Element : OS.Element_Type)
9     return OS.Set;
10 end Aux;
```

Listing 1655: aux.adb

```

1 package body Aux is
2   function Gen_Singleton_Set
3     (Element : OS.Element_Type)
4     return OS.Set
5   is
6   begin
7     return S : OS.Set := OS.Empty_Set do
8       S.Insert (Element);
9     end return;
10   end Gen_Singleton_Set;
11 end Aux;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_Functions_Limited_Types.Constructor_Functions_2
 MD5: b715ae504c49ed59b7fd5ead4cc7bbb4

Since Ada 2005, we can say:

Listing 1656: show_set_decl.adb

```

1 with Ada.Containers.Ordered_Sets;
2 with Aux;
3
4 procedure Show_Set_Decl is
5
6     package Integer_Sets is new
7         Ada.Containers.Ordered_Sets
8         (Element_Type => Integer);
9     use Integer_Sets;
10
11     function Singleton_Set is new
12         Aux.Gen_Singleton_Set
13         (OS => Integer_Sets);
14
15     This_Set : Set := Empty_Set;
16     That_Set : Set := Singleton_Set
17                 (Element => 42);
18
19 begin
20     null;
21 end Show_Set_Decl;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
 Functions_Limited_Types.Constructor_Functions_2
 MD5: 443fc3390b0f3e5516d91c80f16bed3f

whether or not Set is limited. This_Set : Set := Empty_Set; seems clearer than:

Listing 1657: show_set_decl.adb

```

1 with Ada.Containers.Ordered_Sets;
2
3 procedure Show_Set_Decl is
4
5     package Integer_Sets is new
6         Ada.Containers.Ordered_Sets
7         (Element_Type => Integer);
8     use Integer_Sets;
9
10     This_Set : Set;
11
12 begin
13     null;
14 end Show_Set_Decl;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Constructor_
 Functions_Limited_Types.Constructor_Functions_2
 MD5: e5b6c0e148cfdb1987ab3002ec1f53bd

which might mean "default-initialize to the empty set" or might mean "leave it uninitialized, and we'll initialize it in later".

28.3.11 Return objects

Extended return statements for limited types

Note

This section was originally written by Robert A. Duff and published as [Gem #10: Limited Types in Ada 2005](#)³¹⁰.

Previously, we discussed *extended return statements* (page 759). For most types, extended return statements are no big deal — it's just syntactic sugar. But for limited types, this syntax is almost essential:

Listing 1658: task_construct_error.ads

```

1 package Task_Construct_Error is
2
3     task type Task_Type (Discriminant : Integer);
4
5     function Make_Task (Val : Integer)
6         return Task_Type;
7
8 end Task_Construct_Error;
```

Listing 1659: task_construct_error.adb

```

1 package body Task_Construct_Error is
2
3     task body Task_Type is
4     begin
5         null;
6     end Task_Type;
7
8     function Make_Task (Val : Integer)
9         return Task_Type
10    is
11        Result : Task_Type
12            (Discriminant => Val * 3);
13    begin
14        -- some statements...
15        return Result; -- Illegal!
16    end Make_Task;
17
18 end Task_Construct_Error;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_Statements_Limited_Types.Extended_Return_Limited_Error
 MD5: f55b1c367d2931ece4d352d209fe6b3b

The return statement here is illegal, because Result is local to Make_Task, and returning it would involve a copy, which makes no sense (which is why task types are limited). Since Ada 2005, we can write constructor functions for task types:

Listing 1660: task_construct.ads

```

1 package Task_Construct is
2
3     task type Task_Type (Discriminant : Integer);
```

(continues on next page)

³¹⁰ <https://www.adacore.com/gems/ada-gem-10>

(continued from previous page)

```
4      function Make_Task (Val : Integer)
5          return Task_Type;
6
7
8  end Task_Construct;
```

Listing 1661: task_construct.adb

```
1  package body Task_Construct is
2
3      task body Task_Type is
4      begin
5          null;
6      end Task_Type;
7
8      function Make_Task (Val : Integer)
9          return Task_Type is
10     begin
11         return Result : Task_Type
12             (Discriminant => Val * 3)
13     do
14         -- some statements...
15         null;
16     end return;
17     end Make_Task;
18
19 end Task_Construct;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_Statements_Limited_Types.Extended_Return_Limited
MD5: c91a24f09a76aef1c25d1a55bcbee910

If we call it like this:

Listing 1662: show_task_construct.adb

```
1  with Task_Construct; use Task_Construct;
2
3  procedure Show_Task_Construct is
4      My_Task : Task_Type := Make_Task (Val => 42);
5  begin
6      null;
7  end Show_Task_Construct;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_Statements_Limited_Types.Extended_Return_Limited
MD5: 01809b031a844c829f2ead253864ca75

Result is created *in place* in My_Task. Result is temporarily considered local to Make_Task during the *-- some statements* part, but as soon as Make_Task returns, the task becomes more global. Result and My_Task really are one and the same object.

When returning a task from a function, it is activated after the function returns. The *-- some statements* part had better not try to call one of the task's entries, because that would deadlock. That is, the entry call would wait until the task reaches an accept statement, which will never happen, because the task will never be activated.

Initialization and function return

As mentioned in the previous section, the object of limited type returned by the initialization function is built *in place*. In other words, the return object is built in the object that is the target of the assignment statement.

For example, we can see this when looking at the address of the object *returned* by the Init function, which we call to initialize the limited type Simple_Rec:

Listing 1663: limited_types.ads

```

1 package Limited_Types is
2
3     type Integer_Access is access Integer;
4
5     type Simple_Rec is limited private;
6
7     function Init (I : Integer) return Simple_Rec;
8
9 private
10
11     type Simple_Rec is limited record
12         V : Integer_Access;
13     end record;
14
15 end Limited_Types;
```

Listing 1664: limited_types.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;
3 with System.Address_Image;
4
5 package body Limited_Types is
6
7     function Init (I : Integer) return Simple_Rec
8     is
9     begin
10         return E : Simple_Rec do
11             E.V := new Integer'(I);
12
13             Put_Line ("E'Address (Init): "
14                     & System.Address_Image
15                     (E'Address));
16         end return;
17     end Init;
18
19 end Limited_Types;
```

Listing 1665: show_limited_init.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with System;
3 with System.Address_Image;
4
5 with Limited_Types;         use Limited_Types;
6
7 procedure Show_Limited_Init is
8 begin
9     declare
10         A : Simple_Rec := Init (0);
11     begin
```

(continues on next page)

(continued from previous page)

```

12     Put_Line ("A'Address (local): "
13               & System.Address_Image
14               (A'Address));
15 end;
16 Put_Line ("----");
17
18 declare
19     B : Simple_Rec := Init (0);
20 begin
21     Put_Line ("B'Address (local): "
22               & System.Address_Image
23               (B'Address));
24 end;
25 end Show_Limited_Init;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Extended_Return_Statements_Limited_Types.Initialization_Return_Do
 MD5: 67235f804206e07fa4eba3a45cc1096f

Runtime output

```

E'Address (Init): 00007FFE2B2132F8
A'Address (local): 00007FFE2B2132F8
----
E'Address (Init): 00007FFE2B2132F0
B'Address (local): 00007FFE2B2132F0

```

When running this code example and comparing the address of the object E in the Init function and the object that is being initialized in the Show_Limited_Init procedure, we see that the return object E (of the Init function) and the local object in the Show_Limited_Init procedure are the same object.

Important

When we use nonlimited types, we're actually copying the returned object — which was locally created in the function — to the object that we're assigning the function to.

For example, let's modify the previous code and make Simple_Rec nonlimited:

Listing 1666: non_limited_types.ads

```

1  package Non_Limited_Types is
2
3      type Integer_Access is access Integer;
4
5      type Simple_Rec is private;
6
7      function Init (I : Integer)
8                    return Simple_Rec;
9
10 private
11
12     type Simple_Rec is record
13       V : Integer_Access;
14     end record;
15
16 end Non_Limited_Types;

```

Listing 1667: non_limited_types.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with System;
3  with System.Address_Image;
4
5  package body Non_Limited_Types is
6
7      function Init (I : Integer)
8          return Simple_Rec is
9      begin
10         return E : Simple_Rec do
11             E.V := new Integer'(I);
12
13             Put_Line ("E'Address (Init): "
14                     & System.Address_Image
15                     (E'Address));
16         end return;
17     end Init;
18
19 end Non_Limited_Types;
```

Listing 1668: show_non_limited_init_by_copy.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with System;
3  with System.Address_Image;
4
5  with Non_Limited_Types;
6  use Non_Limited_Types;
7
8  procedure Show_Non_Limited_Init_By_Copy is
9      A, B : Simple_Rec;
10 begin
11     declare
12         A : Simple_Rec := Init (0);
13     begin
14         Put_Line ("A'Address (local): "
15                 & System.Address_Image
16                 (A'Address));
17     end;
18     Put_Line ("----");
19
20     declare
21         B : Simple_Rec := Init (0);
22     begin
23         Put_Line ("B'Address (local): "
24                 & System.Address_Image
25                 (B'Address));
26     end;
27 end Show_Non_Limited_Init_By_Copy;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Extended_Return_Statements_Limited_Types.
↳ Initialization_Return_Copy
MD5: 6e224b64b90dabdf5064c70364fa80cb

Runtime output

```

E'Address (Init):  00007FFD80308F30
A'Address (local): 00007FFD80309028
----
E'Address (Init):  00007FFD80308F30
B'Address (local): 00007FFD80309020
```

In this case, we see that the local object E in the Init function is not the same as the object it's being assigned to in the Show_Non_Limited_Init_By_Copy procedure. In fact, E is being copied to A and B.

28.3.12 Building objects from constructors

Note

This section was originally written by Robert A. Duff and published as [Gem #11: Limited Types in Ada 2005](#)³¹¹.

We've earlier seen examples of constructor functions for limited types similar to this:

Listing 1669: p.ads

```
1 with Ada.Strings.Unbounded;
2 use Ada.Strings.Unbounded;
3
4 package P is
5     task type Some_Task_Type;
6
7     protected type Some_Protected_Type is
8         -- dummy type
9     end Some_Protected_Type;
10
11     type T is limited private;
12     function Make_T (Name : String) return T;
13     -- ^^^^^
14     -- constructor function
15 private
16     type T is limited
17         record
18             Name      : Unbounded_String;
19             My_Task    : Some_Task_Type;
20             My_Prot    : Some_Protected_Type;
21         end record;
22 end P;
```

Listing 1670: p.adb

```
1 package body P is
2
3     task body Some_Task_Type is
4     begin
5         null;
6     end Some_Task_Type;
7
8     protected body Some_Protected_Type is
9     end Some_Protected_Type;
10
11     function Make_T (Name : String) return T is
12     begin
13         return (Name =>
14                 To_Unbounded_String (Name),
15                 others => <>);
16     end Make_T;
```

(continues on next page)

³¹¹ <https://www.adacore.com/gems/ada-gem-11>

(continued from previous page)

```

17
18 end P;

```

Listing 1671: p-aux.ads

```

1 package P.Aux is
2     function Make_Rumplestiltskin return T;
3 end P.Aux;

```

Listing 1672: p-aux.adb

```

1 package body P.Aux is
2
3     function Make_Rumplestiltskin return T is
4     begin
5         return Make_T (Name => "Rumplestiltskin");
6     end Make_Rumplestiltskin;
7
8 end P.Aux;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
 ↳ From_Constructors.Building_Objcs_From_Constructors
 MD5: 1956721292a82899d244afcd10ff63ed

It is useful to consider the various contexts in which these functions may be called. We've already seen things like:

Listing 1673: show_rumplestiltskin_constructor.adb

```

1 with P;      use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Rumplestiltskin_Constructor is
5     Rumplestiltskin_Is_My_Name : constant T :=
6         Make_Rumplestiltskin;
7 begin
8     null;
9 end Show_Rumplestiltskin_Constructor;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
 ↳ From_Constructors.Building_Objcs_From_Constructors
 MD5: 2fe193516df6452eccece8132660f8e5

in which case the limited object is built directly in a standalone object. This object will be finalized whenever the surrounding scope is left.

We can also do:

Listing 1674: show_parameter_constructor.adb

```

1 with P;      use P;
2 with P.Aux; use P.Aux;
3
4 procedure Show_Parameter_Constructor is
5     procedure Do_Something (X : T) is null;
6 begin

```

(continues on next page)

(continued from previous page)

```
7   Do_Something (X => Make_Rumplestiltskin);
8 end Show_Parameter_Constructor;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↳ From_Constructors.Building_Objcs_From_Constructors
MD5: 61ccaefb4b7cfc42c065aa15543fc13b

Here, the result of the function is built directly in the formal parameter X of Do_Something. X will be finalized as soon as we return from Do_Something.

We can allocate initialized objects on the heap:

Listing 1675: show_heap_constructor.adb

```
1 with P;      use P;
2 with P.Aux;  use P.Aux;
3
4 procedure Show_Heap_Constructor is
5
6   type T_Ref is access all T;
7
8   Global : T_Ref;
9
10  procedure Heap_Alloc is
11    Local : T_Ref;
12    To_Global : Boolean := True;
13  begin
14    Local := new T'(Make_Rumplestiltskin);
15    if To_Global then
16      Global := Local;
17    end if;
18  end Heap_Alloc;
19
20 begin
21   null;
22 end Show_Heap_Constructor;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
↳ From_Constructors.Building_Objcs_From_Constructors
MD5: 8eb794884f1dfbdbedf1bc4369f45cf8

The result of the function is built directly in the heap-allocated object, which will be finalized when the scope of T_Ref is left (long after Heap_Alloc returns).

We can create another limited type with a component of type T, and use an aggregate:

Listing 1676: show_outer_type.adb

```
1 with P;      use P;
2 with P.Aux;  use P.Aux;
3
4 procedure Show_Outer_Type is
5
6   type Outer_Type is limited record
7     This : T;
8     That : T;
9   end record;
10
```

(continues on next page)

(continued from previous page)

```

11   Outer_Obj : Outer_Type :=
12       (This => Make_Rumplestiltskin,
13        That => Make_T (Name => ""));
14
15   begin
16       null;
17   end Show_Outer_Type;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
 ↳ From_Constructors.Building_Objs_From_Constructors
 MD5: 00817649406492b79977d67eb0fd3955

As usual, the function results are built in place, directly in `Outer_Obj`. This and `Outer_Obj`. That, with no copying involved.

The one case where we *cannot* call such constructor functions is in an assignment statement:

Listing 1677: show_illegal_constructor.adb

```

1   with P;      use P;
2   with P.Aux;  use P.Aux;
3
4   procedure Show_Illegal_Constructor is
5       Rumplestiltskin_Is_My_Name : T;
6   begin
7       Rumplestiltskin_Is_My_Name :=
8           Make_T (Name => ""); -- Illegal!
9   end Show_Illegal_Constructor;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Building_Objects_
 ↳ From_Constructors.Building_Objs_From_Constructors
 MD5: f7b0c78e9fbe2e104b82dfff25ac3e3a

Build output

```

show_illegal_constructor.adb:7:04: error: left hand of assignment must not be
↳ limited type
gprbuild: *** compilation phase failed

```

which is illegal because assignment statements involve copying. Likewise, we can't copy a limited object into some other object:

Listing 1678: show_illegal_constructor.adb

```

1   with P;      use P;
2   with P.Aux;  use P.Aux;
3
4   procedure Show_Illegal_Constructor is
5       Rumplestiltskin_Is_My_Name : constant T :=
6           Make_T (Name => "");
7       Other : T :=
8           Rumplestiltskin_Is_My_Name; -- Illegal!
9   begin
10       null;
11   end Show_Illegal_Constructor;

```

28.3.13 Limited types as parameter

Previously, we saw that *parameters can be passed by copy or by reference* (page 761). Also, we discussed the concept of by-copy and by-reference types. *Explicitly limited types* (page 1079) are by-reference types. Consequently, parameters of these types are always passed by reference.

For further reading...

As an example of the importance of this rule, consider the case of a lock (as an abstract data type). If such a lock object were passed by copy, the Acquire and Release operations would be working on copies of this object, not on the original one. This would lead to timing-dependent bugs.

Let's reuse an example of an explicitly limited type:

Listing 1679: simple_rec.s.ads

```

1 package Simple_Recs is
2
3   type Rec is limited record
4     I : Integer;
5   end record;
6
7 end Simple_Recs;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↪Parameters.Explicitly_Limited_Types
 MD5: de73a20140628420830ed9fe0b2dedb5

In this example, Rec is a by-reference type because the type declaration is an explicit limited record. Therefore, the parameter R of the Proc procedure is passed by reference.

We can run the Test application below and compare the address of the R object from Test to the address of the R parameter of Proc to determine whether both R s refer to the same object or not:

Listing 1680: simple_rec.s.ads

```

1 with System;
2
3 package Simple_Recs is
4
5   type Rec is limited record
6     I : Integer;
7   end record;
8
9   procedure Proc (R : in out Rec;
10                  A : out System.Address);
11
12 end Simple_Recs;
```

Listing 1681: simple_rec.s.adb

```

1 package body Simple_Recs is
2
3   procedure Proc (R : in out Rec;
4                  A : out System.Address) is
```

(continues on next page)

(continued from previous page)

```

5   begin
6       R.I := 0;
7       A  := R'Address;
8   end Proc;
9
10  end Simple_Recs;

```

Listing 1682: test.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with System;                use System;
3  with System.Address_Image;
4  with Simple_Recs;           use Simple_Recs;
5
6  procedure Test is
7      R : Rec;
8
9      AR_Proc, AR_Test : System.Address;
10  begin
11      AR_Proc := R'Address;
12
13      Proc (R, AR_Test);
14
15      Put_Line ("R'Address (Proc): "
16               & System.Address_Image (AR_Proc));
17      Put_Line ("R'Address (Test): "
18               & System.Address_Image (AR_Test));
19
20      if AR_Proc = AR_Test then
21          Put_Line ("R was passed by reference.");
22      else
23          Put_Line ("R was passed by copy.");
24      end if;
25
26  end Test;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Limited_Types.Limited_Types_
 ↪Parameters.Explicitly_Limited_Types
 MD5: d4fe2bb47d2223ef013d22aa305403e5

Runtime output

```

R'Address (Proc): 00007FFCD325377C
R'Address (Test): 00007FFCD325377C
R was passed by reference.

```

When running the Test application, we confirm that R was passed by reference. Note, however, that the fact that R was passed by reference doesn't automatically imply that Rec is a by-reference type: the type could have been ambiguous, and the compiler could have just decided to pass the parameter by reference in this case.

Therefore, we have to rely on the rules specified in the Ada Reference Manual:

1. If a limited type is explicitly limited, a parameter of this type is a by-reference type.
 - The rule applies to all kinds of explicitly limited types. For example, consider private limited types where the type is declared limited in the private type's completion (in the package's private part): a parameter of this type is a by-reference type.

2. If a limited type is not *explicitly* limited, a parameter of this type is neither a by-copy nor a by-reference type.
 - In this case, the decision whether the parameter is passed by reference or by copy is made by the compiler.

i In the Ada Reference Manual

- 6.2 Formal Parameter Modes³¹²
- 6.4.1 Parameter Associations³¹³
- 7.5 Limited Types³¹⁴

28.4 Controlled Types

28.4.1 Overview

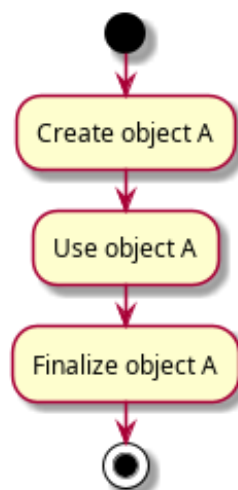
In this section, we introduce the concept of controlled types. We start with a review of lifetime of objects and discuss how controlled types allow us to control the initialization, post-copy (e.g. assignment) adjustment and finalization of objects.

i Relevant topics

- Assignment and Finalization³¹⁵

Lifetime of objects

We already talked about the [lifetime of objects](#)³¹⁶ previously in the context of [access types](#) (page 933). Again, we assume you understand the concept. In any case, let's quickly review the typical lifetime of an object:



In simple terms, an object A is first created before we can make use of it. When object A is

³¹² <http://www.ada-auth.org/standards/22rm/html/RM-6-2.html>

³¹³ <http://www.ada-auth.org/standards/22rm/html/RM-6-4-1.html>

³¹⁴ <http://www.ada-auth.org/standards/22rm/html/RM-7-5.html>

³¹⁵ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

³¹⁶ [https://en.wikipedia.org/wiki/Variable_\(computer_science\)#Scope_and_extent](https://en.wikipedia.org/wiki/Variable_(computer_science)#Scope_and_extent)

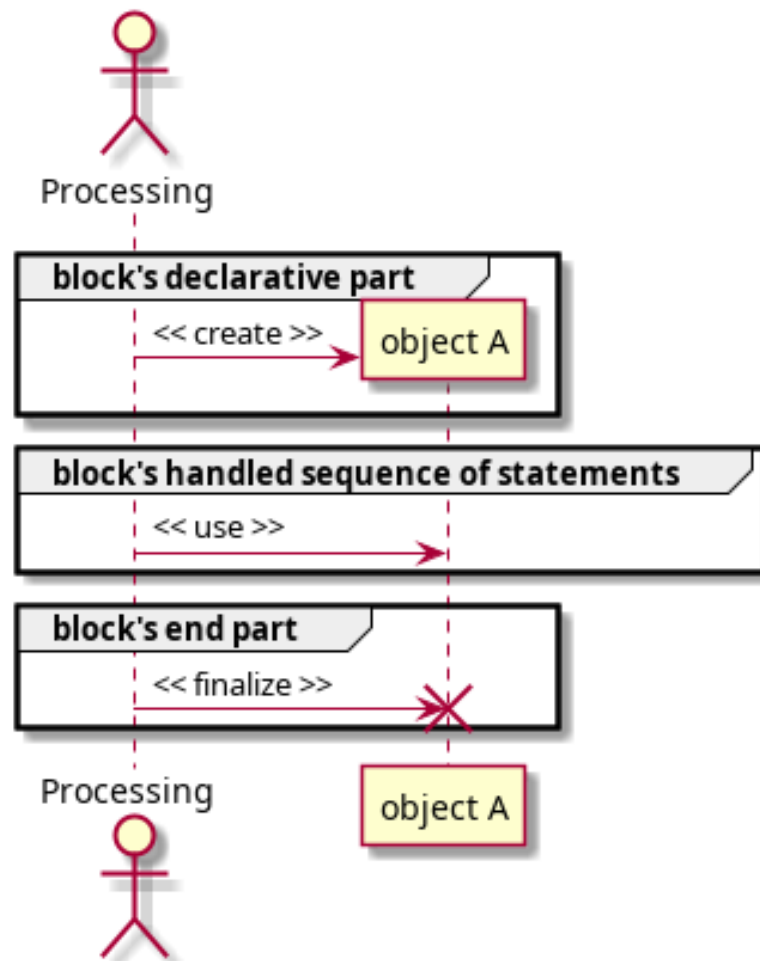
about to get out of scope, it is finalized. Note that finalization might not entail any actual code execution — but it often does.

Let's analyze the lifetime of object A in a procedure P:

```

procedure P is
  A : T;
begin
    P2 (A);
end P;
  
```

We could visualize the lifetime as follows:



In other words, object A is created in the declarative part of P and then it's used in P's sequence of statements. Finally, A is finalized when P ends.

Initialization of objects

Typically, right after an object A is created, it is still uninitialized. Therefore, we have to explicitly initialize it with a meaningful initial value — or with the value returned by a function call, for example. Similarly, when an object A is about to get out of scope, it is going to be finalized (i.e. destroyed) and its contents are then lost forever.

As we know, for some standard Ada types, objects are initialized by default. For example, objects of access types are initialized by default to **null**. Likewise, we can declare *types with default initial value* (page 339):

Listing 1683: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Int is new Integer
5     with Default_Value => 42;
6
7   I : Int;
8   AI : access Int;
9 begin
10   Put_Line ("I : "
11           & I'Image);
12   Put_Line ("AI : "
13           & AI'Image);
14 end Main;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.
↳ Default_Initialization
MD5: 14a5929f0635f0f7843c883bab9021d8

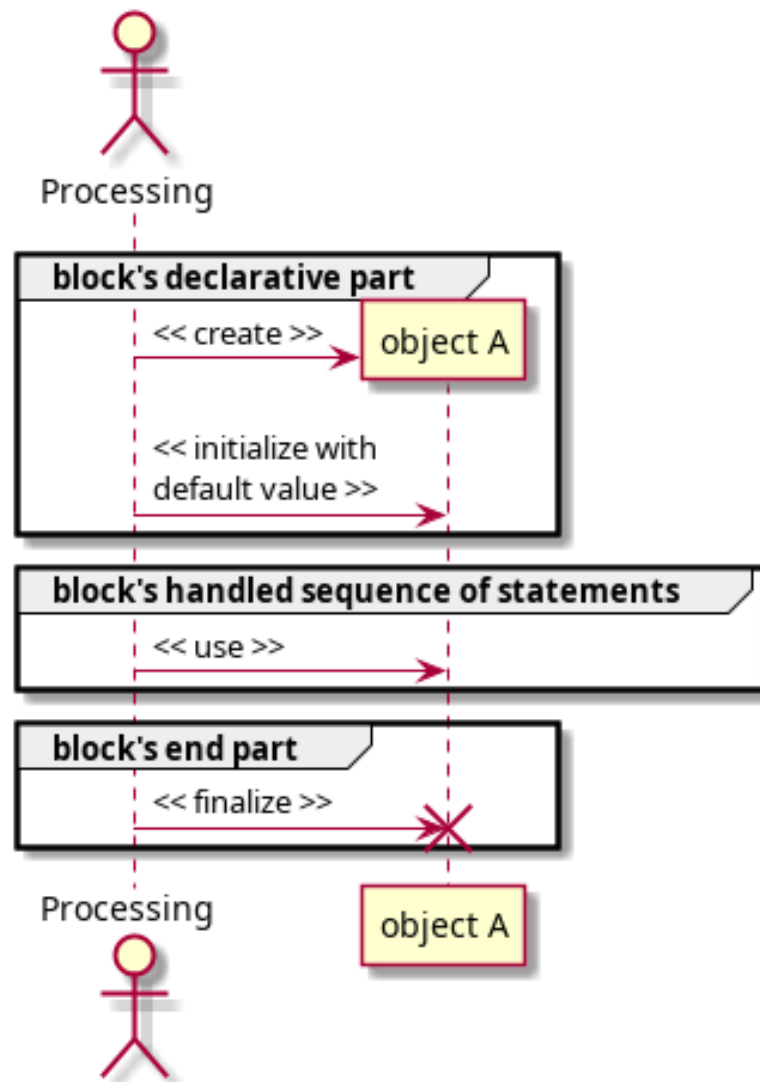
Build output

main.adb:8:04: warning: variable "AI" is read but never assigned [-gnatwv]

Runtime output

I : 42
AI : null

In this case, we can visualize the lifetime of those objects as follows:



Even though these default initialization methods provide some control over the objects, they might not be enough in certain situations. Also, we don't have any means to perform useful operations right before an object gets out of scope.

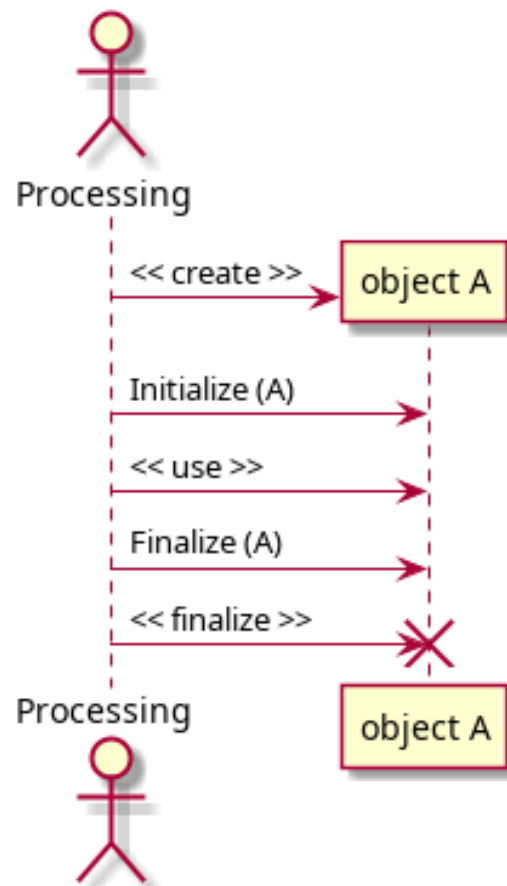
i For further reading...

In general, record types have a very good default initialization capability. They're the most common completion for private types, so the facility is often used. In this sense, default initialization is the first choice, as it's guaranteed and requires nothing of the client. In addition, it's cheap at run-time compared to controlled types.

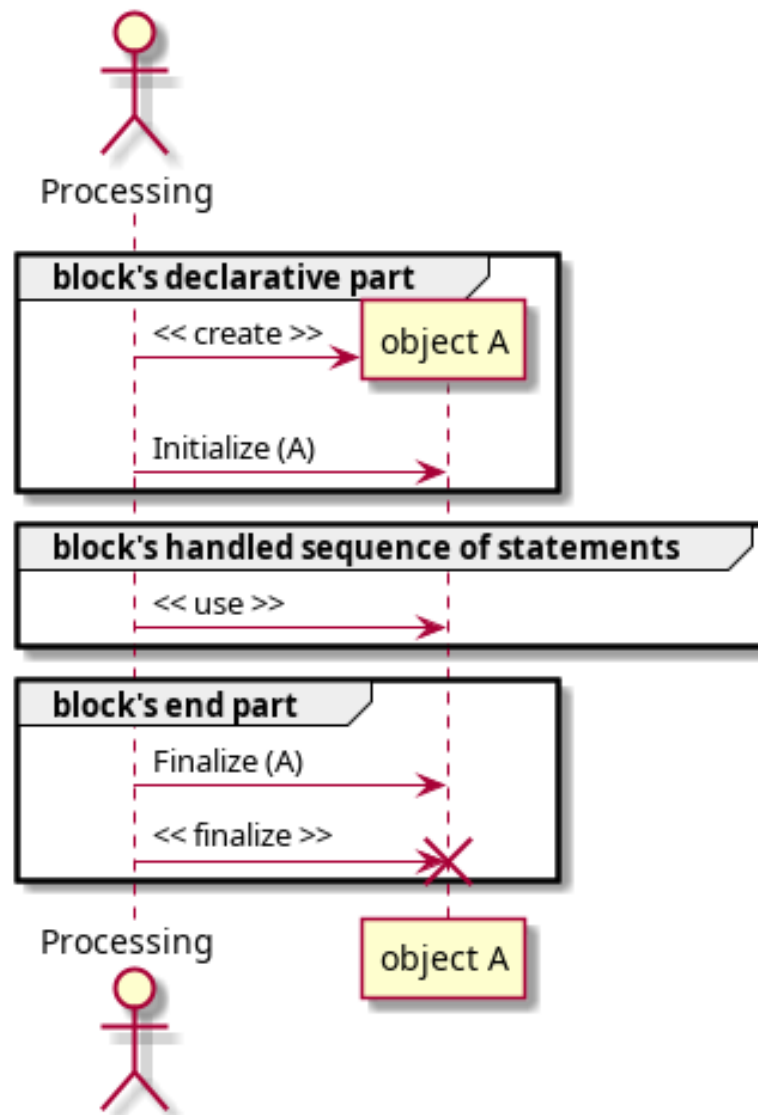
Controlled objects

Controlled objects allow us to better control the initialization and finalization of an object. For any controlled object A, an `Initialize (A)` procedure is called right *after* the object is created, and a `Finalize (A)` procedure is called right *before* the object is actually finalized.

We can visualize the lifetime of controlled objects as follows:



In the context of a block statement, the lifetime becomes:



Let's look at a simple example:

Listing 1684: simple_controlled_types.ads

```

1 with Ada.Finalization;
2
3 package Simple_Controlled_Types is
4
5     type T is tagged private;
6
7     procedure Dummy (E : T);
8
9 private
10
11     type T is new
12         Ada.Finalization.Controlled
13         with null record;
14
15     overriding
16     procedure Initialize (E : in out T);
17
18     overriding

```

(continues on next page)

(continued from previous page)

```
19   procedure Finalize (E : in out T);
20
21 end Simple_Controlled_Types;
```

Listing 1685: simple_controlled_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Controlled_Types is
4
5   procedure Dummy (E : T) is
6   begin
7     Put_Line ("(Dummy...)");
8   end Dummy;
9
10  procedure Initialize (E : in out T) is
11  begin
12    Put_Line ("Initialize...");
13  end Initialize;
14
15  procedure Finalize (E : in out T) is
16  begin
17    Put_Line ("Finalize...");
18  end Finalize;
19
20 end Simple_Controlled_Types;
```

Listing 1686: show_controlled_types.adb

```
1 with Simple_Controlled_Types;
2 use Simple_Controlled_Types;
3
4 procedure Show_Controlled_Types is
5   A : T;
6   --
7   --   This declaration roughly
8   --   corresponds to:
9   --
10  --   A : T;
11  --   begin
12  --     Initialize (A);
13  --
14  begin
15    Dummy (A);
16
17    --   When A is about to get out of
18    --   scope:
19    --
20    --   Finalize (A);
21    --
22 end Show_Controlled_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_Example
MD5: 24f95418bb8c439648ab9dba9f0c953a

Runtime output

```
Initialize...
(Dummy...)
Finalize...
```

When we run this application, we see the user messages indicating the calls to Initialize and Finalize.

For further reading...

Note that if a controlled object isn't used in the application, the compiler might optimize it out. In this case, procedures Initialize and Finalize won't be called for this object, as it doesn't actually exist. You can see this effect by replacing the call to Dummy (A) in the Show_Controlled_Types procedure by a null statement (**null**).

Adjustment of controlled objects

An assignment is a full bit-wise copy of the entire right-hand side to the entire left-hand side. When copying controlled objects, however, we might need to adjust the target object. This is made possible by overriding the Adjust procedure, which is called right after the copy to an object has been performed. (As we'll see later on, *limited controlled types* (page 1127) do not offer an Adjust procedure.)

The *deep copy*³¹⁷ of objects is a typical example where adjustments are necessary. When we assign an object B to an object A, we're essentially doing a *shallow copy*³¹⁸. If we have references to other objects in the source object B, those references will be copied as well, so both target A and source B will be referring to the same objects. When performing a deep copy, however, we want the information from the dereferenced objects to be copied, not the references themselves. Therefore, we have to first allocate new objects for the target object A and copy the information from the original references — the ones we copied from the source object B — to the new objects. This kind of processing can be performed in the Adjust procedure.

As an example, let's extend the previous code example and override the Adjust procedure:

Listing 1687: simple_controlled_types.ads

```
1 with Ada.Finalization;
2
3 package Simple_Controlled_Types is
4
5     type T is tagged private;
6
7     procedure Dummy (E : T);
8
9 private
10
11     type T is new
12         Ada.Finalization.Controlled
13         with null record;
14
15     overriding
16     procedure Initialize (E : in out T);
17
18     overriding
19     procedure Adjust (E : in out T);
20
```

(continues on next page)

³¹⁷ https://en.wikipedia.org/wiki/Object_copying#Deep_copy

³¹⁸ https://en.wikipedia.org/wiki/Object_copying#Shallow_copy

(continued from previous page)

```
21   overriding
22   procedure Finalize (E : in out T);
23
24 end Simple_Controlled_Types;
```

Listing 1688: simple_controlled_types.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      procedure Dummy (E : T) is
6      begin
7          Put_Line ("(Dummy...)");
8      end Dummy;
9
10     procedure Initialize (E : in out T) is
11     begin
12         Put_Line ("Initialize...");
13     end Initialize;
14
15     procedure Adjust (E : in out T) is
16     begin
17         Put_Line ("Adjust...");
18     end Adjust;
19
20     procedure Finalize (E : in out T) is
21     begin
22         Put_Line ("Finalize...");
23     end Finalize;
24
25 end Simple_Controlled_Types;
```

Listing 1689: show_controlled_types.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Simple_Controlled_Types;
4  use Simple_Controlled_Types;
5
6  procedure Show_Controlled_Types is
7      A, B : T;
8  begin
9      Put_Line ("A := B");
10     A := B;
11
12     Dummy (A);
13     Dummy (B);
14 end Show_Controlled_Types;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_Example_2
MD5: 4f4575dab6c9b384ea0cbd8bf9701850

Runtime output

```
Initialize...
Initialize...
```

(continues on next page)

(continued from previous page)

```

A := B
Finalize...
Adjust...
(Dummy...)
(Dummy...)
Finalize...
Finalize...

```

When running this application, we see that the `Adjust` procedure is called for object `A` — right after `B` is copied to `A` as part of the `A := B` assignment. We discuss more about this procedure [later on](#) (page 1141).

Limited controlled types

Ada offers controlled types in two flavors: nonlimited controlled types — such as the ones we've seen so far — and limited controlled types. Both types are declared in the `Ada.Finalization` package.

The only difference between these types is that limited controlled types don't have an `Adjust` procedure that could be overridden, as limited types *do not permit direct copies of objects to be made via assignments* (page 1066). (Obviously, both controlled and limited controlled types provide `Initialize` and `Finalize` procedures.)

The following table summarizes the information:

Type	Name	Initialize	Finalize	Adjust
Nonlimited Controlled	Controlled	Yes	Yes	Yes
Limited controlled	<code>Limited_Controlled</code>	Yes	Yes	Not available

Simple Example with ID

Although the previous code examples indicated that `Initialize`, `Finalize` and `Adjust` are called as we expect for controlled objects, they didn't show us exactly how those objects are actually handled. In this section, we discuss this by analyzing a code example that assigns a unique ID to each controlled object.

Let's start with the complete code example:

Listing 1690: `simple_controlled_types.ads`

```

1 with Ada.Finalization;
2
3 package Simple_Controlled_Types is
4
5     type T is tagged private;
6
7     procedure Show (E      : T;
8                     Name : String);
9
10 private
11
12     protected Id_Gen is
13         procedure New_Id (Id_Out : out Positive);
14     private
15         Id : Natural := 0;
16     end Id_Gen;
17
18     type T is new

```

(continues on next page)

(continued from previous page)

```

19     Ada.Finalization.Controlled with
20     record
21         Id : Positive;
22     end record;
23
24     overriding
25     procedure Initialize (E : in out T);
26
27     overriding
28     procedure Adjust (E : in out T);
29
30     overriding
31     procedure Finalize (E : in out T);
32
33 end Simple_Controlled_Types;

```

Listing 1691: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      protected body Id_Gen is
6
7          procedure New_Id (Id_Out : out Positive) is
8              begin
9                  Id := Id + 1;
10                 Id_Out := Id;
11             end New_Id;
12
13         end Id_Gen;
14
15         procedure Initialize (E : in out T) is
16             begin
17                 Id_Gen.New_Id (E.Id);
18                 Put_Line ("Initialize: ID => "
19                     & E.Id'Image);
20             end Initialize;
21
22         procedure Adjust (E : in out T) is
23             Prev_Id : constant Positive := E.Id;
24             begin
25                 Id_Gen.New_Id (E.Id);
26                 Put_Line ("Adjust:      ID => "
27                     & E.Id'Image);
28                 Put_Line ("      (Previous ID => "
29                     & Prev_Id'Image
30                     & ")");
31             end Adjust;
32
33         procedure Finalize (E : in out T) is
34             begin
35                 Put_Line ("Finalize:   ID => "
36                     & E.Id'Image);
37             end Finalize;
38
39         procedure Show (E      : T;
40                        Name    : String) is
41             begin
42                 Put_Line ("Obj. " & Name
43                     & ": ID => "

```

(continues on next page)

(continued from previous page)

```

44         & E.Id'Image);
45     end Show;
46
47 end Simple_Controlled_Types;

```

Listing 1692: show_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Simple_Controlled_Types;
4  use Simple_Controlled_Types;
5
6  procedure Show_Controlled_Types is
7      A, B : T;
8      --
9      -- Declaration corresponds to:
10     --
11     -- declare
12     --     A, B : T;
13     -- begin
14     --     Initialize (A);
15     --     Initialize (B);
16     -- end;
17 begin
18     Put_Line ("-----");
19     Show (A, "A");
20     Show (B, "B");
21
22     Put_Line ("-----");
23     Put_Line ("A := B;");
24
25     A := B;
26     -- Statement corresponds to:
27     --
28     -- Finalize (A);
29     -- A := B;
30     -- Adjust (A);
31
32     Put_Line ("-----");
33     Show (A, "A");
34     Show (B, "B");
35     Put_Line ("-----");
36
37     -- When A and B get out of scope::
38     --
39     -- Finalize (A);
40     -- Finalize (B);
41     --
42 end Show_Controlled_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Overview.Simple_Example_With_Id
MD5: f7b490041616a1b309184086ceef1b24

Runtime output

```

Initialize: ID => 1
Initialize: ID => 2
-----

```

(continues on next page)

(continued from previous page)

```

Obj. A: ID => 1
Obj. B: ID => 2
-----
A := B;
Finalize:  ID => 1
Adjust:     ID => 3
            (Previous ID => 2)
-----
Obj. A: ID => 3
Obj. B: ID => 2
-----
Finalize:  ID => 2
Finalize:  ID => 3

```

In contrast to the previous versions of the `Simple_Controlled_Types` package, type `T` now has an `Id` component. Moreover, we use a protected object `Id_Gen` that provides us with a unique ID to keep track of each controlled object. Basically, we assign an ID to each controlled object (right after it is created) via the call to `Initialize`. Similarly, this ID is updated via the calls to `Adjust`. Besides, we now have a `Show` procedure that displays the ID of a controlled object.

When running the application, we see that the calls to `Initialize`, `Adjust` and `Finalize` happen as expected. In addition, we see the objects' ID, which we will now analyze in order to understand how each object is actually handled.

First, we see the two calls to `Initialize` for objects `A` and `B`. Object `A`'s ID is 1, and object `B`'s ID is 2. This is later confirmed by the calls to `Show`.

The `A := B` assignment triggers two procedure calls: a call to `Finalize (A)` and a call to `Adjust (A)`. In fact, this assignment can be described as follows:

1. `Finalize (A)` is called before the actual copy;
2. `B`'s data is copied to object `A`;
3. `Adjust (A)` is called after that copy.

We can confirm this via the object ID: the object we handle in the call to `Finalize (A)` has an ID of 1, and the object we handle in the call to `Adjust (A)` has an ID of 2 (which originates from the copy of `B` to `A`) and is later changed (*adjusted*) to 3. Again, we can verify the correct IDs by looking at the output of the calls to `Show`.

Note that the call to `Finalize (A)` (before the copy of `B`'s data) indicates that the previous version of object `A` is being finalized, i.e. it's as though the original object `A` is going to be destroyed and its contents are going to be lost. Actually, the object's contents are just overwritten, but the call to `Finalize` allows us to make proper adjustments to the object before the previous information is lost.

Finally, the new version of object `A` (the one whose ID is 3) and object `B` are finalized via the calls to `Finalize (A)` and `Finalize (B)` before the `Show_Controlled_Types` procedure ends.

28.4.2 Initialization

In this section, we cover some details about the initialization of controlled types. Most of those details are related to the initialization order. In principle, as stated in the Ada Reference Manual, "Initialize and other initialization operations are done in an arbitrary order," except in the situations that we describe later on.

Relevant topics

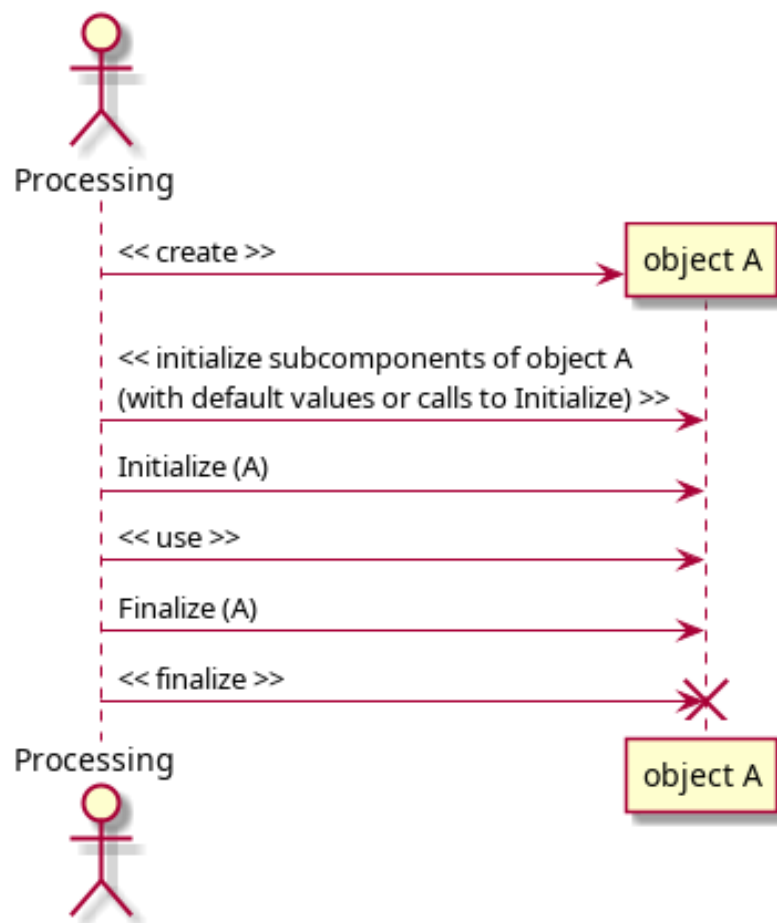
- Assignment and Finalization³¹⁹

Subcomponents

We've seen before that default initialization is a way of controlling the initialization of arbitrary types. In the case of controlled types, the default initialization of its subcomponents always takes place before the call to `Initialize`.

Similarly, a controlled type might have subcomponents of controlled types. These subcomponents are initialized by a call to the `Initialize` procedure of each of those controlled types.

We can visualize the lifetime as follows:



In order to see this effect, let's start by implementing two controlled types: `Sub_1` and `Sub_2`:

Listing 1693: `subs.ads`

```

1 with Ada.Finalization;
2
3 package Subs is
4
5     type Sub_1 is tagged private;
6

```

(continues on next page)

³¹⁹ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

(continued from previous page)

```

7   type Sub_2 is tagged private;
8
9   private
10
11   type Sub_1 is new
12     Ada.Finalization.Controlled
13     with null record;
14
15   overriding
16   procedure Initialize (E : in out Sub_1);
17
18   type Sub_2 is new
19     Ada.Finalization.Controlled
20     with null record;
21
22   overriding
23   procedure Initialize (E : in out Sub_2);
24
25 end Subs;

```

Listing 1694: subs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Subs is
4
5   procedure Initialize (E : in out Sub_1) is
6   begin
7     Put_Line ("Initialize: Sub_1...");
8   end Initialize;
9
10  procedure Initialize (E : in out Sub_2) is
11  begin
12    Put_Line ("Initialize: Sub_2...");
13  end Initialize;
14
15 end Subs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
 ↪Controlled_Initialization
 MD5: f6a7676e82294a62965157d2ffd4ae3b

Now, let's use those controlled types as components of a type T. In addition, let's declare an integer component I with default initialization. This is how the complete code looks like:

Listing 1695: simple_controlled_types.ads

```

1 with Ada.Finalization;
2
3 with Subs; use Subs;
4
5 package Simple_Controlled_Types is
6
7   type T is tagged private;
8
9   procedure Dummy (E : T);
10
11 private
12

```

(continues on next page)

(continued from previous page)

```

13  function Default_Init return Integer;
14
15  type T is new
16    Ada.Finalization.Controlled with
17    record
18      S1 : Sub_1;
19      S2 : Sub_2;
20      I  : Integer := Default_Init;
21    end record;
22
23  overriding
24  procedure Initialize (E : in out T);
25
26  end Simple_Controlled_Types;

```

Listing 1696: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      function Default_Init return Integer is
6      begin
7          Put_Line ("Default_Init: Integer...");
8          return 42;
9      end Default_Init;
10
11     procedure Dummy (E : T) is
12     begin
13         Put_Line ("(Dummy: T...)");
14     end Dummy;
15
16     procedure Initialize (E : in out T) is
17     begin
18         Put_Line ("Initialize: T...");
19     end Initialize;
20
21 end Simple_Controlled_Types;

```

Listing 1697: show_controlled_types.adb

```

1  with Simple_Controlled_Types;
2  use Simple_Controlled_Types;
3
4  procedure Show_Controlled_Types is
5      A : T;
6  begin
7      Dummy (A);
8  end Show_Controlled_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
 ↪ Controlled_Initialization
 MD5: 39d0efa76c056ac8190573c86f17c890

Runtime output

```

Initialize: Sub_1...
Initialize: Sub_2...

```

(continues on next page)

(continued from previous page)

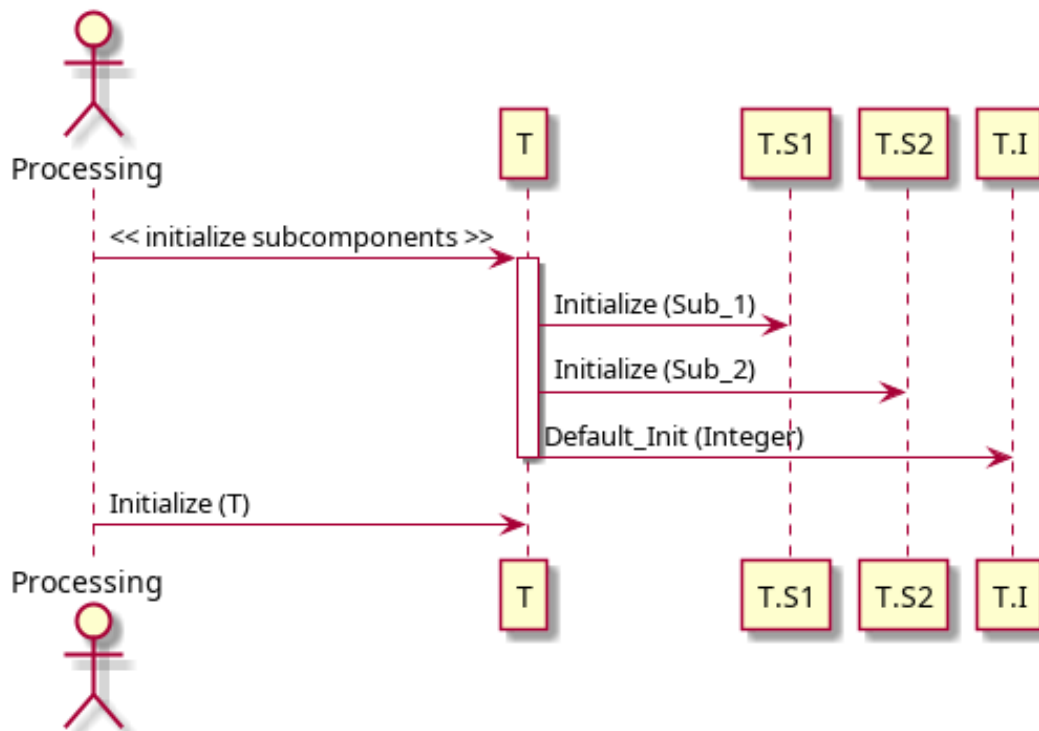
```

Default_Init: Integer...
Initialize: T...
(Dummy: T...)

```

When we run this application, we see that the Sub_1 and Sub_2 components are initialized by calls to their respective Initialize procedures, and the I component is initialized with its default value (via a call to the Default_Init function). Finally, after all subcomponents of type T have been initialized, the Initialize procedure is called for the type T itself.

This diagram shows the initialization sequence:



Components with access discriminants

Record types with access discriminants are a special case. In fact, according to the Ada Reference Manual, "if an object has a component with an access discriminant constrained by a *per-object expression* (page 511), Initialize is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, Initialize is applied to them in order of their component declarations."

Let's see a code example. First, we implement another package with controlled types:

Listing 1698: selections.ads

```

1 with Ada.Finalization;
2
3 package Selections is
4
5     type Selection is private;
6
7     type Selection_1 (S : access Selection) is
8         tagged private;
9
10    type Selection_2 (S : access Selection) is

```

(continues on next page)

(continued from previous page)

```

11     tagged private;
12
13 private
14
15     type Selection is null record;
16
17     type Selection_1 (S : access Selection) is new
18         Ada.Finalization.Controlled
19         with null record;
20
21     overriding
22     procedure Initialize
23         (E : in out Selection_1);
24
25     type Selection_2 (S : access Selection) is new
26         Ada.Finalization.Controlled
27         with null record;
28
29     overriding
30     procedure Initialize
31         (E : in out Selection_2);
32
33 end Selections;

```

Listing 1699: selections.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Selections is
4
5     procedure Initialize
6         (E : in out Selection_1) is
7     begin
8         Put_Line ("Initialize: Selection_1...");
9     end Initialize;
10
11    procedure Initialize
12        (E : in out Selection_2) is
13    begin
14        Put_Line ("Initialize: Selection_2...");
15    end Initialize;
16
17 end Selections;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
 ↪Controlled_Initialization
 MD5: 01c3639ebd52d37856e77ccfeb057d1b

In this example, we see the declaration of the Selection_1 and Selection_2 types, which are controlled types with an access discriminant of Selection type. Now, let's use these types in the declaration of the T type from the [previous example](#) (page 1131) and add two new components (Sel_1 and Sel_2):

Listing 1700: simple_controlled_types.ads

```

1 with Ada.Finalization;
2
3 with Subs;      use Subs;

```

(continues on next page)

(continued from previous page)

```

4  with Selections; use Selections;
5
6  package Simple_Controlled_Types is
7
8      type T (S1 : access Selection;
9              S2 : access Selection) is
10         tagged private;
11
12     procedure Dummy (E : T);
13
14 private
15
16     function Default_Init return Integer;
17
18     type T (S1 : access Selection;
19             S2 : access Selection) is new
20         Ada.Finalization.Controlled with
21     record
22         Sel_1 : Selection_1 (S1);
23         Sel_2 : Selection_2 (S2);
24         S_1   : Sub_1;
25         I     : Integer := Default_Init;
26     end record;
27
28     overriding
29     procedure Initialize (E : in out T);
30
31 end Simple_Controlled_Types;

```

Listing 1701: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      function Default_Init return Integer is
6      begin
7          Put_Line ("Default_Init: Integer...");
8          return 42;
9      end Default_Init;
10
11     procedure Dummy (E : T) is
12     begin
13         Put_Line ("(Dummy: T...)");
14     end Dummy;
15
16     procedure Initialize (E : in out T) is
17     begin
18         Put_Line ("Initialize: T...");
19     end Initialize;
20
21 end Simple_Controlled_Types;

```

Listing 1702: show_controlled_types.adb

```

1  with Simple_Controlled_Types;
2  use Simple_Controlled_Types;
3
4  with Selections;
5  use Selections;

```

(continues on next page)

(continued from previous page)

```

6
7 procedure Show_Controlled_Types is
8     S1, S2 : aliased Selection;
9     A : T (S1'Access, S2'Access);
10 begin
11     Dummy (A);
12 end Show_Controlled_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
 ↪Controlled_Initialization
 MD5: 74f507b912ab746b70aec451a9bc8f74

Runtime output

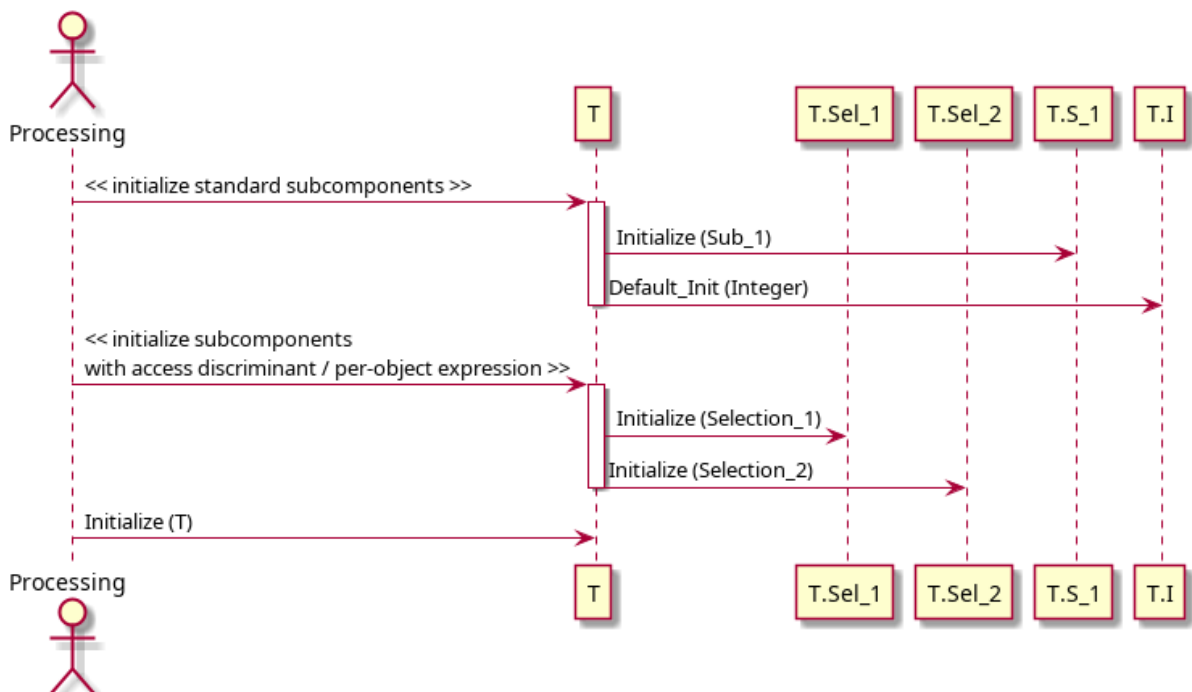
```

Initialize: Sub_1...
Default_Init: Integer...
Initialize: Selection_1...
Initialize: Selection_2...
Initialize: T...
(Dummy: T...)

```

When running this example, we see that all other subcomponents — to be more precise, those subcomponents that require initialization — are initialized before the Sub_1 and Sub_2 components are initialized via calls to their corresponding Initialize procedure. Note that, although Sub_1 and Sub_2 are the last components to be initialized, they are still initialized before the call to the Initialize procedure of type T.

This diagram shows the initialization sequence:



Task activation

Components of task types also require special treatment. According to the Ada Reference Manual, "for an allocator, any task activations follow all calls on Initialize."

As always, let's analyze an example that illustrates this. First, we implement another package called Workers with a simple task type:

Listing 1703: workers.ads

```
1 package Workers is
2
3     task type Worker is
4         entry Start;
5         entry Stop;
6     end Worker;
7
8 end Workers;
```

Listing 1704: workers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Workers is
4
5     task body Worker is
6
7         function Init return Integer is
8         begin
9             Put_Line ("Activating Worker task...");
10            return 0;
11        end Init;
12
13        I : Integer := Init;
14        begin
15
16            accept Start do
17                Put_Line ("Worker.Start accepted...");
18                I := I + 1;
19            end Start;
20
21            accept Stop do
22                Put_Line ("Worker.Stop accepted...");
23                I := I - 1;
24            end Stop;
25        end Worker;
26
27 end Workers;
```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↳ Controlled_Initialization
MD5: 1d48a78f14a496c8cdadeab9d1bc9070

Let's extend the declaration of the T type from the [previous example](#) (page 1134) and declare a new component of Worker type. Note that we have to change T to a limited controlled type because of this new component of task type. This is the updated code:

Listing 1705: simple_controlled_types.ads

```

1  with Ada.Finalization;
2
3  with Subs;          use Subs;
4  with Selections; use Selections;
5  with Workers;      use Workers;
6
7  package Simple_Controlled_Types is
8
9      type T (S : access Selection) is
10         tagged limited private;
11
12     procedure Start_Work (E : T);
13     procedure Stop_Work (E : T);
14
15 private
16
17     function Default_Init return Integer;
18
19     type T (S : access Selection) is new
20         Ada.Finalization.Limited_Controlled with
21         record
22             W      : Worker;
23             Sel_1   : Selection_1 (S);
24             S1      : Sub_1;
25             I       : Integer := Default_Init;
26         end record;
27
28     overriding
29     procedure Initialize (E : in out T);
30
31 end Simple_Controlled_Types;

```

Listing 1706: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      function Default_Init return Integer is
6      begin
7          Put_Line ("Default_Init: Integer...");
8          return 42;
9      end Default_Init;
10
11     procedure Start_Work (E : T) is
12     begin
13         -- Starting Worker task:
14         E.W.Start;
15
16     end Start_Work;
17
18     procedure Stop_Work (E : T) is
19     begin
20         -- Stopping Worker task:
21         E.W.Stop;
22     end Stop_Work;
23
24     procedure Initialize (E : in out T) is
25     begin

```

(continues on next page)

(continued from previous page)

```
26     Put_Line ("Initialize: T...");
27     end Initialize;
28
29 end Simple_Controlled_Types;
```

Listing 1707: show_controlled_types.adb

```
1  with Simple_Controlled_Types;
2  use Simple_Controlled_Types;
3
4  with Selections; use Selections;
5
6  procedure Show_Controlled_Types is
7      type T_Access is access T;
8
9      S : aliased Selection;
10     A : constant T_Access := new T (S'Access);
11 begin
12     Start_Work (A.all);
13     Stop_Work (A.all);
14 end Show_Controlled_Types;
```

Code block metadata

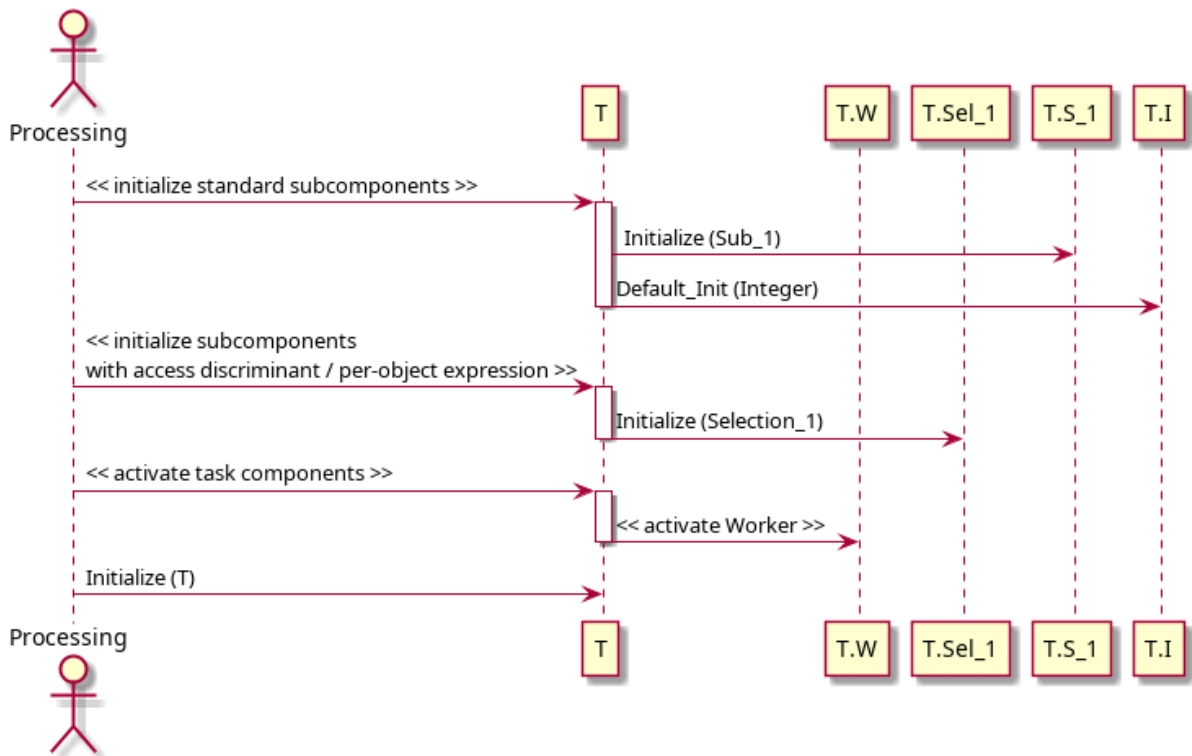
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Initialization.
↳Controlled_Initialization
MD5: f87adac74205d590ee66ce971918e642

Runtime output

```
Initialize: Sub_1...
Default_Init: Integer...
Initialize: Selection_1...
Initialize: T...
Activating Worker task...
Worker.Start accepted...
Worker.Stop accepted...
```

When we run this application, we see that the W component is activated only after all other subcomponents of type T have been initialized.

This diagram shows the initialization sequence:



28.4.3 Assignment

We already talked about *adjustments* (page 1125) previously. As we already mentioned, an actual assignment is a full bit-wise copy of the entire right-hand side to the entire left-hand side, so the adjustment (via a call to `Adjust`) is a way to "work around" that, when necessary. In this section, we'll look into some details about the adjustment of controlled types.

i Relevant topics

- [Assignment and Finalization](#)³²⁰

Assignment using anonymous object

The [Ada Reference Manual](#)³²¹ mentions that an anonymous object is created during the assignment of objects of controlled type. A simple `A := B` operation for nonlimited controlled types can be expanded to the following illustrative code:

```

procedure P is
  A, B: Some_Controlled_Type;
begin
  --
  --  A := B;
  --
  B_To_A_Assignment : declare
    Anon_Obj : Some_Controlled_Type;
  begin
    Anon_Obj := B;
    Adjust (Anon_Obj);
  
```

(continues on next page)

³²⁰ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

³²¹ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

(continued from previous page)

```

    Finalize (A);
    A := Anon_Obj;
    Finalize (Anon_Obj);
end B_To_A_Assignment;
end P;

```

The first assignment happens to the anonymous object `Anon_Obj`. After the adjustment of `Anon_Obj` and the finalization of the original version of `A`, the actual assignment to `A` can take place — and `Anon_Obj` can be discarded after it has been properly finalized. With this strategy, we have a chance to finalize the original version of `A` before the assignment overwrites the object.

Of course, this expanded code isn't really efficient, and the compiler has some freedom to improve the performance of the generated machine code. Whenever possible, it'll typically optimize the anonymous object out and build the object in place. (The [Ada Reference Manual](#)³²² describes the rules when this is possible or not.)

Also, the `A := Anon_Obj` statement in the code above doesn't necessarily translate to an actual assignment in the generated machine code. Typically, a compiler may treat `Anon_Obj` as the new `A` and destroy the original version of `A` (i.e. the object that used to be `A`). In this case, the code becomes something like this:

```

procedure P is
  A, B: Some_Controlled_Type;
begin
  --
  --  A := B;
  --
  B_To_A_Assignment : declare
    Anon_Obj : Some_Controlled_Type;
  begin
    Anon_Obj := B;
    Finalize (A);
    Adjust (Anon_Obj);
  declare
    A : Some_Controlled_Type renames Anon_Obj;
  begin
    --  Now, we treat Anon_Obj as the new A.
    --  Further processing continues here...

  end;
end B_To_A_Assignment;
end P;

```

In some cases, the compiler is required to build the object in place. A typical example is when an object of controlled type is initialized by assigning an aggregate to it:

```

C: constant Some_Controlled_Type :=
  (Ada.Finalization.Controlled with ...);
--  C is built in place,
--  no anonymous object is used here.

```

Also, it's possible that `Adjust` and `Finalize` aren't called at all. Consider an assignment like this: `A := A;`. In this case, since the object on both sides is the same, the compiler is allowed to simply skip the assignment and not do anything.

For more details about possible optimizations and compiler behavior, please refer to the [Ada Reference Manual](#)³²³.

³²² <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

³²³ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

In general, the advice is simple: use `Adjust` and `Finalize` solely for their intended purposes. In other words, don't implement extraneous side-effects into those procedures, as they might not be called at run-time.

Adjustment of subcomponents

In principle, the order in which components are adjusted is arbitrary. However, adjustments of subcomponents will happen before the adjustment of the component itself. The subcomponents must be adjusted before the enclosing object because the semantics of the adjustment of the whole might depend on the states of the parts (the subcomponents), so those states must already be in place.

Let's revisit a [previous code example](#) (page 1131). First, we override the `Adjust` procedure of the `Sub_1` and `Sub_2` types from the `Subs` package.

Listing 1708: subs.ads

```

1  with Ada.Finalization;
2
3  package Subs is
4
5      type Sub_1 is tagged private;
6
7      type Sub_2 is tagged private;
8
9  private
10
11     type Sub_1 is new
12         Ada.Finalization.Controlled
13         with null record;
14
15     overriding
16     procedure Initialize (E : in out Sub_1);
17
18     overriding
19     procedure Adjust (E : in out Sub_1);
20
21     overriding
22     procedure Finalize (E : in out Sub_1);
23
24     type Sub_2 is new
25         Ada.Finalization.Controlled
26         with null record;
27
28     overriding
29     procedure Initialize (E : in out Sub_2);
30
31     overriding
32     procedure Adjust (E : in out Sub_2);
33
34     overriding
35     procedure Finalize (E : in out Sub_2);
36
37 end Subs;
```

Listing 1709: subs.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Subs is
4
```

(continues on next page)

(continued from previous page)

```

5  procedure Initialize (E : in out Sub_1) is
6  begin
7      Put_Line ("Initialize: Sub_1...");
8  end Initialize;
9
10 procedure Adjust (E : in out Sub_1) is
11 begin
12     Put_Line ("Adjust: Sub_1...");
13 end Adjust;
14
15 procedure Finalize (E : in out Sub_1) is
16 begin
17     Put_Line ("Finalize: Sub_1...");
18 end Finalize;
19
20 procedure Initialize (E : in out Sub_2) is
21 begin
22     Put_Line ("Initialize: Sub_2...");
23 end Initialize;
24
25 procedure Adjust (E : in out Sub_2) is
26 begin
27     Put_Line ("Adjust: Sub_2...");
28 end Adjust;
29
30 procedure Finalize (E : in out Sub_2) is
31 begin
32     Put_Line ("Finalize: Sub_2...");
33 end Finalize;
34
35 end Subs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
↳ Controlled_Initialization
MD5: 110d88543a7a897ba433c90f6c2a881c

Next, we override the Adjust procedure of the T type from the Simple_Controlled_Types package:

Listing 1710: simple_controlled_types.ads

```

1  with Ada.Finalization;
2
3  with Subs; use Subs;
4
5  package Simple_Controlled_Types is
6
7      type T is tagged private;
8
9      procedure Dummy (E : T);
10
11 private
12
13     function Default_Init return Integer;
14
15     type T is new
16         Ada.Finalization.Controlled with
17     record
18         S1 : Sub_1;

```

(continues on next page)

(continued from previous page)

```

19     S2 : Sub_2;
20     I  : Integer := Default_Init;
21 end record;
22
23 overriding
24 procedure Initialize (E : in out T);
25
26 overriding
27 procedure Adjust (E : in out T);
28
29 overriding
30 procedure Finalize (E : in out T);
31
32 end Simple_Controlled_Types;

```

Listing 1711: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      function Default_Init return Integer is
6      begin
7          Put_Line ("Default_Init: Integer...");
8          return 42;
9      end Default_Init;
10
11     procedure Dummy (E : T) is
12     begin
13         Put_Line ("(Dummy: T...)");
14     end Dummy;
15
16     procedure Initialize (E : in out T) is
17     begin
18         Put_Line ("Initialize: T...");
19     end Initialize;
20
21     procedure Adjust (E : in out T) is
22     begin
23         Put_Line ("Adjust: T...");
24     end Adjust;
25
26     procedure Finalize (E : in out T) is
27     begin
28         Put_Line ("Finalize: T...");
29     end Finalize;
30
31 end Simple_Controlled_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
 ↪ Controlled_Initialization
 MD5: 9fb392305df70734994cffe612cb3869

Finally, this is the main application:

Listing 1712: show_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```
3 with Simple_Controlled_Types;
4 use Simple_Controlled_Types;
5
6 procedure Show_Controlled_Types is
7   A, B : T;
8 begin
9   Dummy (A);
10
11   Put_Line ("-----");
12   Put_Line ("A := B");
13   A := B;
14   Put_Line ("-----");
15 end Show_Controlled_Types;
```

Code block metadata

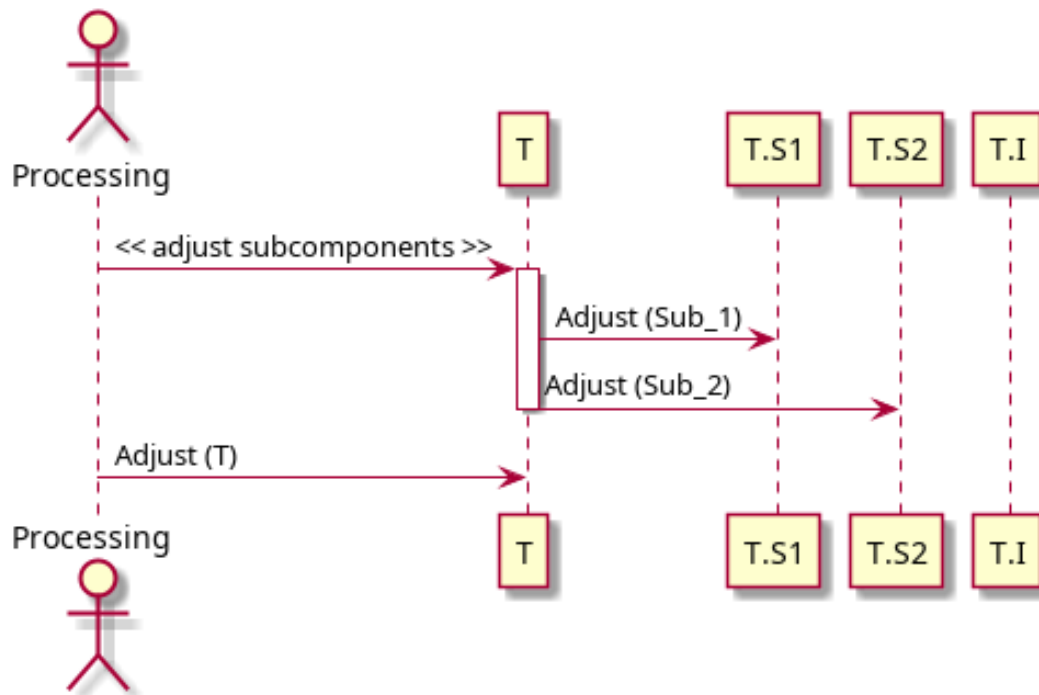
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Adjustment.
↳ Controlled_Initialization
MD5: 1ceaa50cbb18b9f1f997246a614e3a90

Runtime output

```
Initialize: Sub_1...
Initialize: Sub_2...
Default_Init: Integer...
Initialize: T...
Initialize: Sub_1...
Initialize: Sub_2...
Default_Init: Integer...
Initialize: T...
(Dummy: T...)
-----
A := B
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
Adjust: Sub_1...
Adjust: Sub_2...
Adjust: T...
-----
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
```

When running this code, we see that the S1 and S2 components are adjusted before the adjustment of the parent type T takes place.

This diagram shows the adjustment sequence:



28.4.4 Finalization

We mentioned finalization — and the `Finalize` procedure — at the *beginning of the chapter* (page 1121). In this section, we discuss the topic in more detail.

Relevant topics

- [Assignment and Finalization](#)³²⁴
- [Completion and Finalization](#)³²⁵

Normal and abnormal completion

When a subprogram has just executed its last statement, normal completion of this subprogram has been reached. At this point, finalization starts. In the case of controlled objects, this means that the `Finalize` procedure is called for those objects. (As we've already seen *an example of normal completion* (page 1123) at the beginning of the chapter, we won't repeat it here, as we assume you are already familiar with the concept.)

When an exception is raised or due to an abort, however, a subprogram has an abnormal completion. We discuss more about exception handling and finalization *later on* (page 1156).

Finalization via unchecked deallocation

When performing unchecked deallocation of a controlled type, the `Finalize` procedure is called right before the actual memory for the controlled object is deallocated.

Let's see a simple example:

³²⁴ <http://www.ada-auth.org/standards/22rm/html/RM-7-6.html>

³²⁵ <http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html>

Listing 1713: simple_controlled_types.ads

```
1 with Ada.Finalization;
2 with Ada.Unchecked_Deallocation;
3
4 package Simple_Controlled_Types is
5
6     type T is tagged private;
7
8     procedure Dummy (E : T);
9
10    type T_Access is access T;
11
12    procedure Free (A : in out T_Access);
13
14 private
15
16    type T is new
17        Ada.Finalization.Controlled
18        with null record;
19
20    overriding
21    procedure Finalize (E : in out T);
22
23    procedure Free_T_Access is
24        new Ada.Unchecked_Deallocation
25        (Object => T,
26         Name  => T_Access);
27
28    procedure Free (A : in out T_Access)
29        renames Free_T_Access;
30
31 end Simple_Controlled_Types;
```

Listing 1714: simple_controlled_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Controlled_Types is
4
5     procedure Dummy (E : T) is
6     begin
7         Put_Line ("(Dummy T...)");
8     end Dummy;
9
10    procedure Finalize (E : in out T) is
11    begin
12        Put_Line ("Finalize T...");
13    end Finalize;
14
15 end Simple_Controlled_Types;
```

Listing 1715: show_controlled_types.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Simple_Controlled_Types;
4 use Simple_Controlled_Types;
5
6 procedure Show_Controlled_Types is
7     A : T_Access := new T;
```

(continues on next page)

(continued from previous page)

```

8 begin
9   Dummy (A.all);
10
11   Free (A);
12   -- At this point, Finalize (A.all)
13   -- will be called before the actual
14   -- deallocation.
15
16   Put_Line ("We've just freed A.");
17 end Show_Controlled_Types;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↳Unchecked_Deallocation
MD5: b9388699ee396430f689fe88df41fc32

```

Runtime output

```

(Dummy T...)
Finalize T...
We've just freed A.

```

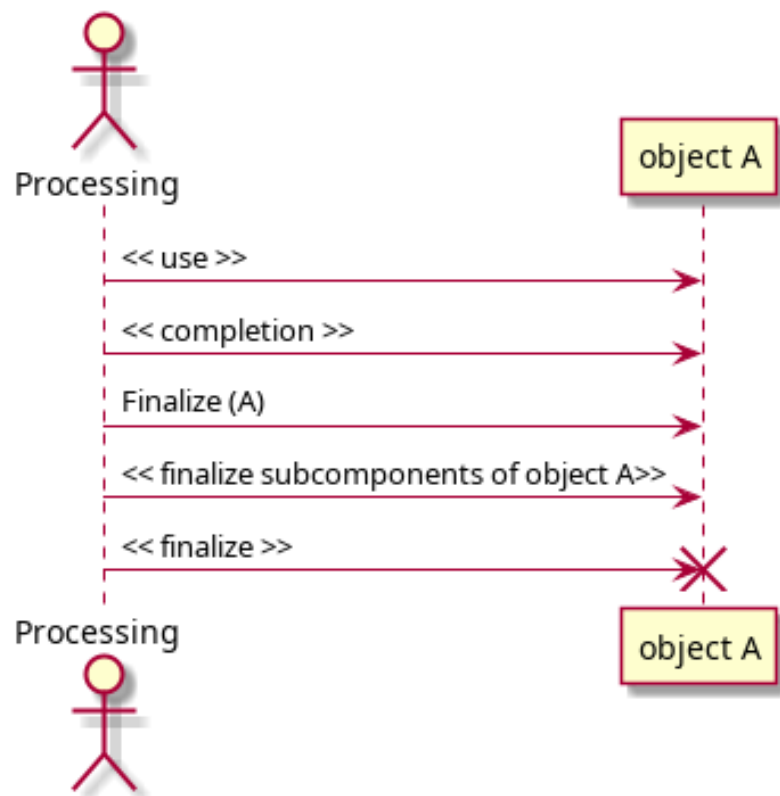
In this example, we see that a call to `Finalize` (for type `T`) is triggered by the call to `Free` for the `A` object — at this point, we haven't reached the end of the main procedure (`Show_Controlled_Types`) yet. After the call to `Free`, the object originally referenced by `A` has been completely finalized — and deallocated.

When the main procedure completes (after the call to `Put_Line` in that procedure), we would normally see the calls to `Finalize` for controlled objects. However, at this point, we obviously don't have a second call to the `Finalize` procedure for type `T`, as the object referenced by `A` has already been finalized and freed.

Subcomponents

As we've seen in the section about *initialization of subcomponents* (page 1131), subcomponents of a controlled type are initialized by a call to their corresponding `Initialize` procedure before the call to `Initialize` for the parent controlled type. In the case of finalization, the reverse order is applied: first, finalization of the parent type takes place, and then the finalization of the subcomponents.

We can visualize the lifetime as follows:



Let's show a code example by revisiting the previous implementation of the controlled types `Sub_1` and `Sub_2`, and adapting it:

Listing 1716: subs.ads

```

1  with Ada.Finalization;
2
3  package Subs is
4
5      type Sub_1 is tagged private;
6
7      type Sub_2 is tagged private;
8
9  private
10
11     type Sub_1 is new
12       Ada.Finalization.Controlled
13       with null record;
14
15     overriding
16     procedure Finalize (E : in out Sub_1);
17
18     type Sub_2 is new
19       Ada.Finalization.Controlled
20       with null record;
21
22     overriding
23     procedure Finalize (E : in out Sub_2);
24
25  end Subs;
  
```

Listing 1717: subs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Subs is
4
5     procedure Finalize (E : in out Sub_1) is
6     begin
7         Put_Line ("Finalize: Sub_1...");
8     end Finalize;
9
10    procedure Finalize (E : in out Sub_2) is
11    begin
12        Put_Line ("Finalize: Sub_2...");
13    end Finalize;
14
15 end Subs;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
 ↳Controlled_Initialization
 MD5: 565f0b13586c08e0cdfdc119bcb28780

Now, let's use those controlled types as components of a type T:

Listing 1718: simple_controlled_types.ads

```

1 with Ada.Finalization;
2
3 with Subs; use Subs;
4
5 package Simple_Controlled_Types is
6
7     type T is tagged private;
8
9     procedure Dummy (E : T);
10
11 private
12
13     type T is new
14         Ada.Finalization.Controlled with
15         record
16             S1 : Sub_1;
17             S2 : Sub_2;
18         end record;
19
20     overriding
21     procedure Finalize (E : in out T);
22
23 end Simple_Controlled_Types;

```

Listing 1719: simple_controlled_types.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Simple_Controlled_Types is
4
5     procedure Dummy (E : T) is
6     begin
7         Put_Line ("(Dummy: T...)");

```

(continues on next page)

(continued from previous page)

```
8   end Dummy;
9
10  procedure Finalize (E : in out T) is
11  begin
12      Put_Line ("Finalize: T...");
13  end Finalize;
14
15  end Simple_Controlled_Types;
```

Listing 1720: show_controlled_types.adb

```
1  with Simple_Controlled_Types;
2  use   Simple_Controlled_Types;
3
4  procedure Show_Controlled_Types is
5      A : T;
6  begin
7      Dummy (A);
8  end Show_Controlled_Types;
```

Code block metadata

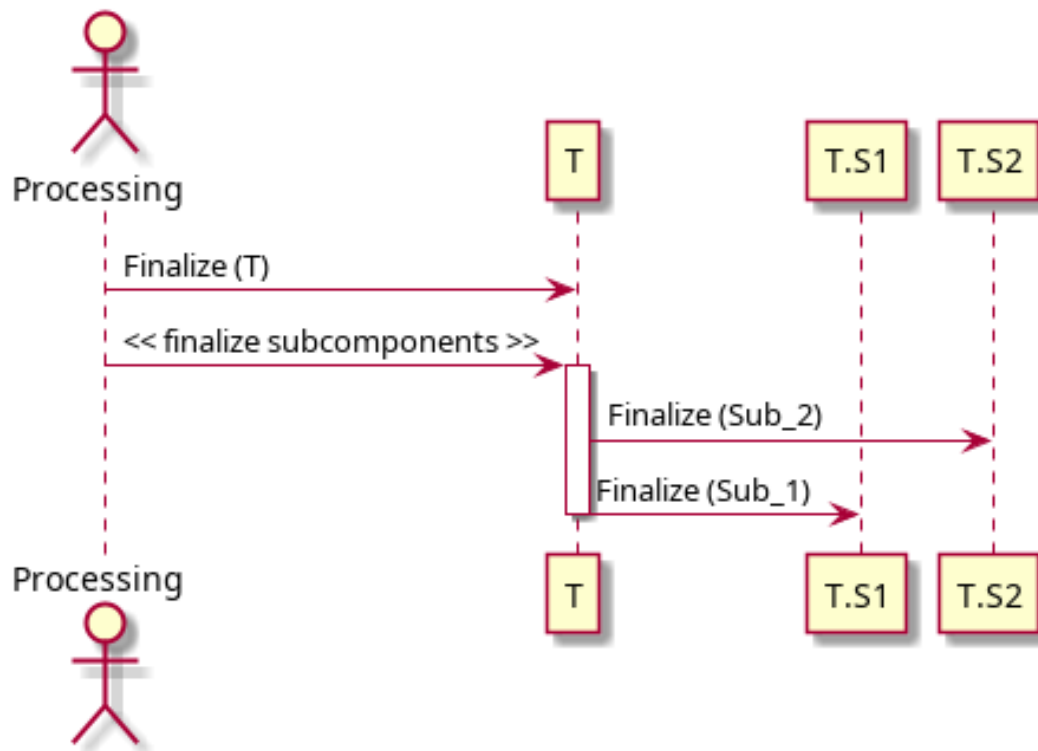
Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↳Controlled_Initialization
MD5: 6feecb7c544f340bf4841034d7ab5f71

Runtime output

```
(Dummy: T...)
Finalize: T...
Finalize: Sub_2...
Finalize: Sub_1...
```

When we run this application, we see that the `Finalize` procedure is called for the type `T` itself — as the first step of the finalization of type `T`. Then, the `Sub_2` and `Sub_1` components are finalized by calls to their respective `Finalize` procedures.

This diagram shows the finalization sequence:



Components with access discriminants

We already discussed the *initialization of components with access discriminants constrained by a per-object expression* (page 1134). In the case of the finalization of such components, they are finalized before any components that do not fall into this category — in the reverse order of their component declarations — but after the finalization of the parent type.

Let's revisit a *previous code example* (page 1134) and adapt it to demonstrate the finalization of components with access discriminants. First, we implement another package with controlled types:

Listing 1721: selections.ads

```

1 with Ada.Finalization;
2
3 package Selections is
4
5     type Selection is private;
6
7     type Selection_1 (S : access Selection) is
8         tagged private;
9
10    type Selection_2 (S : access Selection) is
11        tagged private;
12
13 private
14
15     type Selection is null record;
16
17     type Selection_1 (S : access Selection) is new
18         Ada.Finalization.Controlled
19         with null record;
20
21     overriding
22     procedure Finalize
  
```

(continues on next page)

(continued from previous page)

```

23     (E : in out Selection_1);
24
25     type Selection_2 (S : access Selection) is new
26       Ada.Finalization.Controlled
27       with null record;
28
29     overriding
30     procedure Finalize
31       (E : in out Selection_2);
32
33 end Selections;

```

Listing 1722: selections.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Selections is
4
5      procedure Finalize
6        (E : in out Selection_1) is
7      begin
8          Put_Line ("Finalize: Selection_1...");
9      end Finalize;
10
11     procedure Finalize
12       (E : in out Selection_2) is
13     begin
14         Put_Line ("Finalize: Selection_2...");
15     end Finalize;
16
17 end Selections;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
 ↪Controlled_Initialization
 MD5: d1d35eb7ea62742fb130fbf05d898989

In this example, we see the declaration of the Selection_1 and Selection_2 types, which are controlled types with an access discriminant of Selection type. Now, let's use these types in the declaration of a type T and add two new components — Sel_1 and Sel_2:

Listing 1723: simple_controlled_types.ads

```

1  with Ada.Finalization;
2
3  with Subs;      use Subs;
4  with Selections; use Selections;
5
6  package Simple_Controlled_Types is
7
8      type T (S1 : access Selection;
9              S2 : access Selection) is
10         tagged private;
11
12     procedure Dummy (E : T);
13
14 private
15
16     type T (S1 : access Selection;

```

(continues on next page)

(continued from previous page)

```

17         S2 : access Selection) is new
18         Ada.Finalization.Controlled with
19         record
20             Sel_1 : Selection_1 (S1);
21             Sel_2 : Selection_2 (S2);
22             S_1   : Sub_1;
23         end record;
24
25         overriding
26         procedure Finalize (E : in out T);
27
28     end Simple_Controlled_Types;

```

Listing 1724: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      procedure Dummy (E : T) is
6      begin
7          Put_Line ("(Dummy: T...)");
8      end Dummy;
9
10     procedure Finalize (E : in out T) is
11     begin
12         Put_Line ("Finalize: T...");
13     end Finalize;
14
15 end Simple_Controlled_Types;

```

Listing 1725: show_controlled_types.adb

```

1  with Simple_Controlled_Types;
2  use Simple_Controlled_Types;
3
4  with Selections;
5  use Selections;
6
7  procedure Show_Controlled_Types is
8      S1, S2 : aliased Selection;
9      A : T (S1'Access, S2'Access);
10 begin
11     Dummy (A);
12 end Show_Controlled_Types;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Finalization.
↳ Controlled_Initialization
MD5: e421a750f11ade3b4df98569c71b904a

Runtime output

```

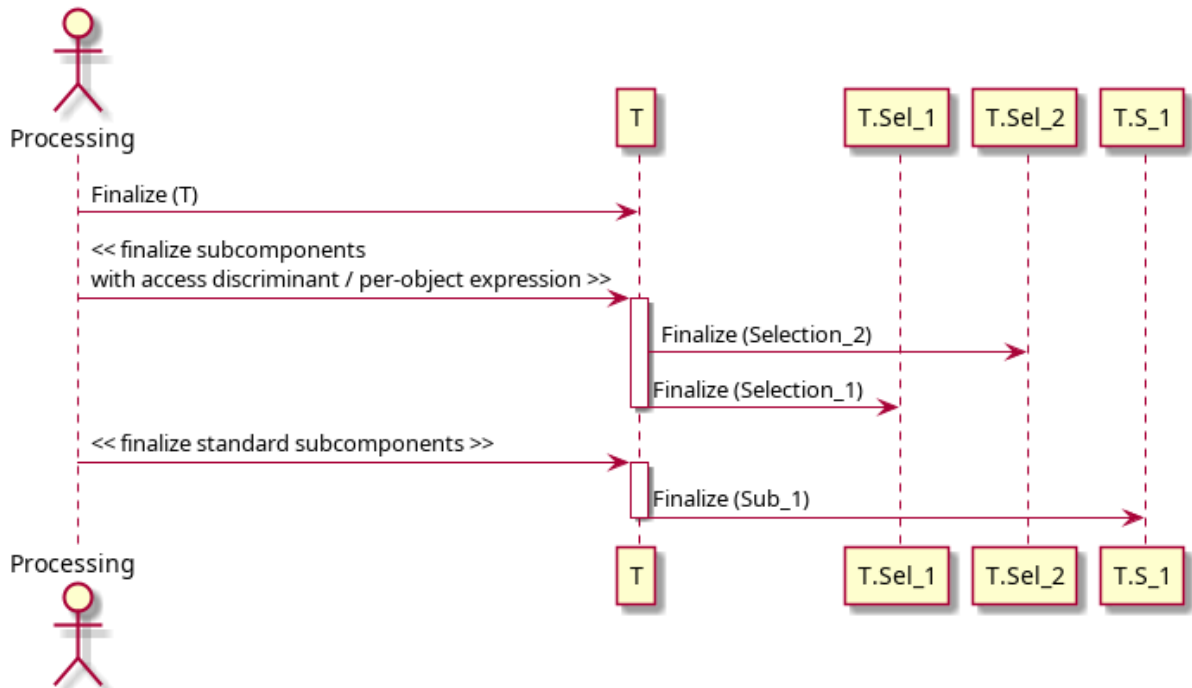
(Dummy: T...)
Finalize: T...
Finalize: Selection_2...
Finalize: Selection_1...
Finalize: Sub_1...

```

When we run this example, we see that the Finalize procedure of type T is called as

the first step. Then, the `Finalize` procedure is called for the components with an access discriminant constrained by a *per-object expression* (page 511) — in this case, `Sel_2` and `Sel_1` (of `Selection_2` and `Selection_1` types, respectively). Finally, the `Sub_1` component is finalized.

This diagram shows the finalization sequence:



28.4.5 Controlled Types and Exception Handling

In the previous section, we mainly focused on the normal completion of controlled types. However, when control is transferred out of the normal execution path due to an abort or an exception being raised, we speak of abnormal completion. In this section, we focus on those cases.

Let's start with a simple example:

Listing 1726: `simple_controlled_types.ads`

```

1 with Ada.Finalization;
2
3 package Simple_Controlled_Types is
4
5     type T is tagged private;
6
7     procedure Dummy (E : T);
8
9 private
10
11     type T is new
12         Ada.Finalization.Controlled
13         with null record;
14
15     overriding
16     procedure Initialize (E : in out T);
17
18     overriding
19     procedure Finalize (E : in out T);
  
```

(continues on next page)

(continued from previous page)

```

20
21 end Simple_Controlled_Types;

```

Listing 1727: simple_controlled_types.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Controlled_Types is
4
5      procedure Dummy (E : T) is
6      begin
7          Put_Line ("(Dummy...)");
8      end Dummy;
9
10     procedure Initialize (E : in out T) is
11     begin
12         Put_Line ("Initialize...");
13     end Initialize;
14
15     procedure Finalize (E : in out T) is
16     begin
17         Put_Line ("Finalize...");
18     end Finalize;
19
20 end Simple_Controlled_Types;

```

Listing 1728: show_simple_exception.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Simple_Controlled_Types;
4  use Simple_Controlled_Types;
5
6  procedure Show_Simple_Exception is
7      A : T;
8
9      function Int_Last return Integer is
10         (Integer'Last);
11
12     Cnt : Positive := Int_Last;
13 begin
14     Cnt := Cnt + 1;
15
16     Dummy (A);
17
18     Put_Line (Cnt'Image);
19
20     -- When A is about to get out of
21     -- scope:
22     --
23     -- Finalize (A);
24     --
25 end Show_Simple_Exception;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_

 ↳ Handling.Simple_Exception
 MD5: 9461f420f091f058e6ealee419b2a5c6

Runtime output


```
Initialize...  
Finalize...
```

```
raised CONSTRAINT_ERROR : show_simple_exception.adb:14 overflow check failed
```

In this example, we're forcing an overflow to happen in the `Show_Simple_Exception` by adding one to the integer variable `Cnt`, which already has the value `Integer'Last`. The corresponding *overflow check* (page 813) raises the `Constraint_Error`.

However, *before* this exception is raised, the finalization of the controlled object `A` is performed. In this sense, we have normal completion of the controlled type — even though an exception is being raised.

For further reading...

We already talked about the *allocation check* (page 817), which may raise a `Program_Error` exception. In the code example for that section, we used controlled types. Feel free to revisit the example.

Relevant topics

- [Completion and Finalization](#)³²⁶

Exception raising in Initialize

If an exception is raised in the `Initialize` procedure, we have abnormal completion. Let's see an example:

Listing 1729: `ct_initialize_exception.ads`

```
1 with Ada.Finalization;  
2  
3 package CT_Initialize_Exception is  
4  
5     type T is tagged private;  
6  
7     procedure Dummy (E : T);  
8  
9 private  
10  
11     type T is new  
12         Ada.Finalization.Controlled  
13         with null record;  
14  
15     overriding  
16     procedure Initialize (E : in out T);  
17  
18     overriding  
19     procedure Finalize (E : in out T);  
20  
21 end CT_Initialize_Exception;
```

³²⁶ <http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html>

Listing 1730: ct_initialize_exception.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body CT_Initialize_Exception is
4
5     function Int_Last return Integer is
6         (Integer'Last);
7
8     Cnt : Positive := Int_Last;
9
10    procedure Dummy (E : T) is
11    begin
12        Put_Line ("(Dummy...)");
13    end Dummy;
14
15    procedure Initialize (E : in out T) is
16    begin
17        Put_Line ("Initialize...");
18        Cnt := Cnt + 1;
19    end Initialize;
20
21    procedure Finalize (E : in out T) is
22    begin
23        Put_Line ("Finalize...");
24    end Finalize;
25
26 end CT_Initialize_Exception;

```

Listing 1731: show_initialize_exception.adb

```

1 with CT_Initialize_Exception;
2 use CT_Initialize_Exception;
3
4 procedure Show_Initialize_Exception is
5     A : T;
6 begin
7     Dummy (A);
8 end Show_Initialize_Exception;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
 ↳ Handling.CT_Initialize_Exception
 MD5: 189a5fafafb01eba31a73c9237fa7aff

Runtime output

```

Initialize...

raised CONSTRAINT_ERROR : ct_initialize_exception.adb:18 overflow check failed

```

In the `Show_Initialize_Exception` procedure, we declare an object `A` of controlled type `T`. As we know, this declaration triggers a call to the `Initialize` procedure that we've implemented in the body of the `CT_Initialize_Exception` package. In the `Initialize` procedure, we're forcing an overflow to happen — by adding one to the `Cnt` variable, which already has the `Integer'Last` value.

This is an example of abnormal completion, as the control is transferred out of the `Initialize` procedure, and the corresponding `Finalize` procedure is never called for object `A`.

Bounded errors of controlled types

Bounded errors (page 800) are an important topic when talking about exception and controlled types. In general, if an exception is raised in the `Adjust` or `Finalize` procedure, this is considered a bounded error. If the bounded error is detected, the `Program_Error` exception is raised.

Note that the original exception raised in the `Adjust` or `Finalize` procedures could be any possible exception. For example, one of those procedures could raise a `Constraint_Error` exception. However, the actual exception that is raised at runtime is the `Program_Error` exception. This is because the bounded error, which raises the `Program_Error` exception, is more severe than the original exception coming from those procedures.

(The behavior is different when the `Adjust` or `Finalize` procedure is called explicitly, as we'll see later.)

Not every exception raised during an operation on controlled types is considered a bounded error. In fact, the case we've seen before, an *exception raised in the `Initialize` procedure* (page 1158) is not a bounded error.

Here's a code example of a `Constraint_Error` exception being raised in the `Finalize` procedure:

Listing 1732: `ct_finalize_exception.ads`

```

1  with Ada.Finalization;
2
3  package CT_Finalize_Exception is
4
5      type T is tagged private;
6
7      procedure Dummy (E : T);
8
9      procedure Reset_Counter;
10
11  private
12
13      type T is new
14          Ada.Finalization.Controlled
15              with null record;
16
17      overriding
18      procedure Initialize (E : in out T);
19
20      overriding
21      procedure Adjust (E : in out T);
22
23      overriding
24      procedure Finalize (E : in out T);
25
26  end CT_Finalize_Exception;
```

Listing 1733: `ct_finalize_exception.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body CT_Finalize_Exception is
4
5      Cnt : Integer := Integer'Last;
6
7      procedure Dummy (E : T) is
8      begin
9          Put_Line ("(Dummy...)");
```

(continues on next page)

(continued from previous page)

```

10  end Dummy;
11
12  procedure Initialize (E : in out T) is
13  begin
14      Put_Line ("Initialize...");
15  end Initialize;
16
17  overriding
18  procedure Adjust (E : in out T) is
19  begin
20      Put_Line ("Adjust...");
21  end Adjust;
22
23  procedure Finalize (E : in out T) is
24  begin
25      Put_Line ("Finalize...");
26      Cnt := Cnt + 1;
27  end Finalize;
28
29  procedure Reset_Counter is
30  begin
31      Cnt := 0;
32  end Reset_Counter;
33
34  end CT_Finalize_Exception;

```

Listing 1734: show_finalize_exception.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with CT_Finalize_Exception;
4  use CT_Finalize_Exception;
5
6  procedure Show_Finalize_Exception is
7      A, B : T;
8  begin
9      Dummy (A);
10
11      -- When A is about to get out of
12      -- scope:
13      --
14      -- Finalize (A);
15      --
16  end Show_Finalize_Exception;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
 ↳ Handling.CT_Finalize_Exception
 MD5: eacb64b0a9d68ce3484a3bda9b633495

Runtime output

```

Initialize...
Initialize...
(Dummy...)
Finalize...
Finalize...

raised PROGRAM_ERROR : show_finalize_exception.adb:6 finalize/adjust raised_
↳ exception

```

In this example, we're again forcing an overflow to happen (by adding one to the integer variable `Cnt`), this time in the `Finalize` procedure. When this procedure is implicitly called — when object `A` is about to get out of scope in the `Show_Finalize_Exception` procedure — the `Constraint_Error` exception is raised.

As we've just seen, having an exception be raised during an implicit call to the `Finalize` procedure is a bounded error. Therefore, we see that the `Program_Error` exception is raised at runtime instead of the original `Constraint_Error` exception.

As we hinted in the beginning, when the `Adjust` or the `Finalize` procedure is called *explicitly*, the exception raised in that procedure is *not* considered a bounded error. In this case, the original exception is raised.

To show an example of such an explicit call, let's first move the overridden procedures for type `T` (`Initialize`, `Adjust` and `Finalize`) out of the private part of the package `CT_Finalize_Exception`, so they are now visible to clients. This allows us to call the `Finalize` procedure explicitly:

Listing 1735: `ct_finalize_exception.ads`

```
1 with Ada.Finalization;
2
3 package CT_Finalize_Exception is
4
5     type T is new
6         Ada.Finalization.Controlled
7         with null record;
8
9     overriding
10    procedure Initialize (E : in out T);
11
12    overriding
13    procedure Adjust (E : in out T);
14
15    overriding
16    procedure Finalize (E : in out T);
17
18    procedure Dummy (E : T);
19
20    procedure Reset_Counter;
21
22 end CT_Finalize_Exception;
```

Listing 1736: `show_finalize_exception.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with CT_Finalize_Exception;
4 use CT_Finalize_Exception;
5
6 procedure Show_Finalize_Exception is
7     A : T;
8 begin
9     Dummy (A);
10
11     Finalize (A);
12
13     Put_Line ("After Finalize");
14 exception
15     when Constraint_Error =>
16         Put_Line
17             ("Constraint_Error is being handled...");
```

(continues on next page)

(continued from previous page)

```

18     Reset_Counter;
19 end Show_Finalize_Exception;

```

Code block metadata

```

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Exception_
        Handling.CT_Finalize_Exception
MD5: f43133c997076c491a20117960be8807

```

Runtime output

```

Initialize...
(Dummy...)
Finalize...
Constraint_Error is being handled...
Finalize...

```

Now, we're calling the `Finalize` procedure explicitly in the `Show_Finalize_Exception` procedure. As we know, due to the operation on `I` in the `Finalize` procedure, the `Constraint_Error` exception is raised in the procedure. Because we're handling this exception in the `Show_Finalize_Exception` procedure, we see the corresponding user message ("Constraint_Error is being handled...") at runtime.

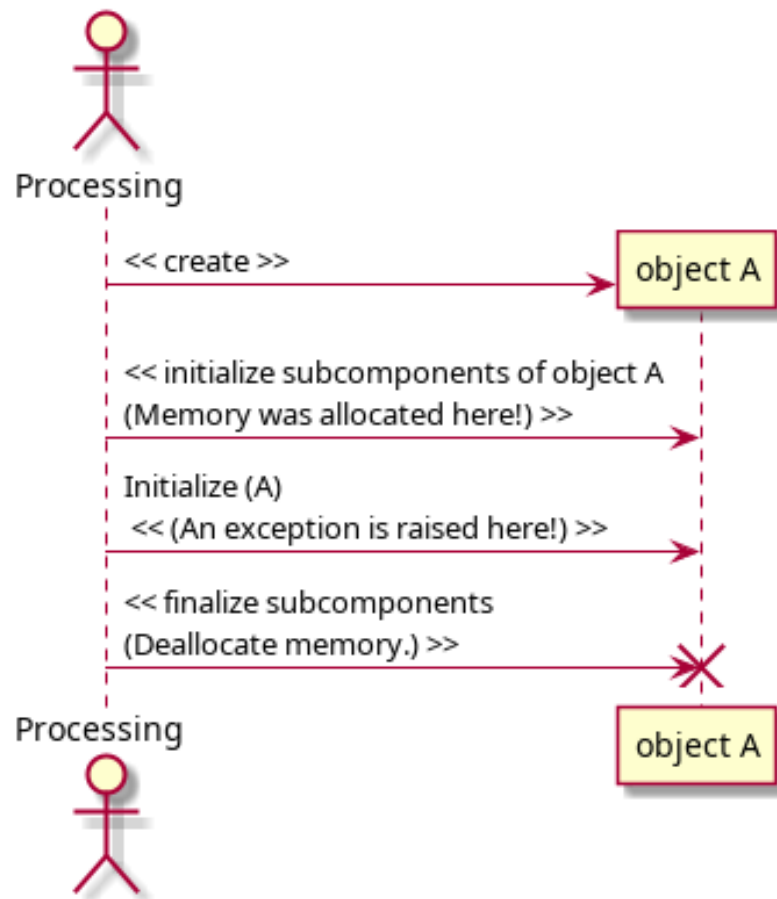
(Note that in the exception handling block, we're calling the `Reset_Counter` procedure. This prevents `Constraint_Error` from being raised in the next call to `Finalize`.)

Memory allocation and exceptions

When a memory block is allocated for controlled types and a bounded error occurs, there is no guarantee that this memory block will be deallocated. Roughly speaking, the compiler has the freedom — but not the obligation — to generate appropriate calls to `Finalize`, which may deallocate memory blocks.

For example, we've seen that *subcomponents of controlled type* (page 1131) of a controlled object `A` are initialized before the initialization of object `A` takes place. Because memory might have been allocated for the subcomponents, the compiler can insert code that attempts to finalize those subcomponents, which in turn deallocates the memory blocks (if they were allocated in the first place).

We can visualize this strategy in the following diagram:



This strategy (of finalizing subcomponents that haven't raised exceptions) prevents memory leaks. However, this behavior very much depends on the compiler implementation. The [Ada Reference Manual](#)³²⁷ delineates (in the "Implementation Permissions" section) the cases where the compiler is allowed — but not required — to finalize objects when exceptions are raised.

Because the actual behavior isn't defined, custom implementation of `Adjust` and `Finalize` procedures for controlled types should be designed very carefully in order to avoid exceptions, especially when memory is allocated in the `Initialize` procedure.

28.4.6 Applications of Controlled Types

In this section, we discuss applications of controlled types. In this context, it's important to remember that controlled types have an associated overhead, which can become non-negligible depending in which context the controlled objects are used. However, there are applications where utilizing controlled types is the best approach.

(Note that this overhead we've just mentioned is not specific to Ada. In fact, types similar to controlled types will be relatively expensive in any programming language. As an example, destructors in C++ may require a similar maintenance of state at run-time.)

Encapsulating access type handling

Previously, when discussing *design strategies for access types* (page 956), we saw an example on using *limited controlled types to encapsulate access types* (page 960).

³²⁷ <http://www.ada-auth.org/standards/22rm/html/RM-7-6-1.html>

A more generalized example is the one of an unbounded stack. Because it's unbounded, it allows for increasing the stack's size *on demand*. We can implement this kind of stack by using access types. Let's look at a simple (unoptimized) implementation:

Listing 1737: unbounded_stacks.ads

```

1  with Ada.Finalization;
2
3  generic
4      Default_Chunk_Size : Positive := 5;
5      type Element is private;
6  package Unbounded_Stacks is
7
8      Stack_Underflow : exception;
9
10     type Unbounded_Stack is private;
11
12     procedure Push (S : in out Unbounded_Stack;
13                   E : Element);
14
15     function Pop (S : in out Unbounded_Stack)
16                 return Element;
17
18     function Is_Empty (S : Unbounded_Stack)
19                     return Boolean;
20
21 private
22
23     type Element_Array is
24         array (Positive range <>) of
25             Element;
26
27     type Element_Array_Access is
28         access Element_Array;
29
30     type Unbounded_Stack is new
31         Ada.Finalization.Controlled with
32         record
33             Chunk_Size : Positive
34                 := Default_Chunk_Size;
35             Data       : Element_Array_Access;
36             Top         : Natural := 0;
37         end record;
38
39     procedure Initialize
40         (S : in out Unbounded_Stack);
41
42     procedure Adjust
43         (S : in out Unbounded_Stack);
44
45     procedure Finalize
46         (S : in out Unbounded_Stack);
47
48 end Unbounded_Stacks;

```

Listing 1738: unbounded_stacks.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Ada.Unchecked_Deallocation;
4
5  package body Unbounded_Stacks is

```

(continues on next page)

(continued from previous page)

```

6
7
8  --  LOCAL SUBPROGRAMS
9
10
11 procedure Free is
12   new Ada.Unchecked_Deallocation
13     (Object => Element_Array,
14      Name   => Element_Array_Access);
15
16 function Is_Full (S : Unbounded_Stack)
17   return Boolean is
18 begin
19   return S.Top = S.Data'Last;
20 end Is_Full;
21
22 procedure Reallocate_Data
23   (To      : in out Element_Array_Access;
24    From    :      Element_Array_Access;
25    Max_Last :      Positive;
26    Valid_Last :      Positive) is
27 begin
28   To := new Element_Array (1 .. Max_Last);
29
30   for I in 1 .. Valid_Last loop
31     To (I) := From (I);
32   end loop;
33 end Reallocate_Data;
34
35 procedure Increase_Size
36   (S : in out Unbounded_Stack)
37 is
38   Old_Data : Element_Array_Access := S.Data;
39   Old_Last : constant Positive
40     := Old_Data'Last;
41   New_Last : constant Positive
42     := Old_Data'Last + S.Chunk_Size;
43 begin
44   Put_Line ("Increasing Unbounded_Stack "
45     & "(1 .. "
46     & Old_Last'Image
47     & ") to (1 .. "
48     & New_Last'Image
49     & ")");
50
51   Reallocate_Data
52     (To      => S.Data,
53      From    => Old_Data,
54      Max_Last => New_Last,
55      Valid_Last => S.Top);
56
57   Free (Old_Data);
58 end Increase_Size;
59
60
61 --  SUBPROGRAMS
62
63
64 procedure Push (S : in out Unbounded_Stack;
65               E :      Element) is
66 begin

```

(continues on next page)

(continued from previous page)

```

67     if Is_Full (S) then
68         Increase_Size (S);
69     end if;
70
71     S.Top := S.Top + 1;
72     S.Data (S.Top) := E;
73 end Push;
74
75 function Pop (S : in out Unbounded_Stack)
76     return Element is
77 begin
78     return E : Element do
79         if Is_Empty (S) then
80             raise Stack_Underflow;
81         end if;
82
83         E := S.Data (S.Top);
84         S.Top := S.Top - 1;
85     end return;
86 end Pop;
87
88 function Is_Empty (S : Unbounded_Stack)
89     return Boolean is
90 begin
91     return S.Top = 0;
92 end Is_Empty;
93
94 --
95 -- PRIVATE SUBPROGRAMS
96 --
97
98 procedure Initialize
99     (S : in out Unbounded_Stack)
100 is
101     Last : constant Positive
102         := S.Chunk_Size;
103 begin
104     Put_Line ("Initializing Unbounded_Stack "
105             & "(1 .. "
106             & Last'Image
107             & ")");
108     S.Data := new Element_Array
109             (1 .. S.Chunk_Size);
110 end Initialize;
111
112 procedure Allocate_Duplicate_Data
113     (S : in out Unbounded_Stack)
114 is
115     Last : constant Positive
116         := S.Data'Last;
117 begin
118     Put_Line ("Duplicating data for new "
119             & "Unbounded_Stack (1 .. "
120             & Last'Image
121             & ")");
122
123     Reallocate_Data
124         (To      => S.Data,
125          From    => S.Data,
126          Max_Last => Last,
127          Valid_Last => S.Top);

```

(continues on next page)

(continued from previous page)

```

128   end Allocate_Duplicate_Data;
129
130   procedure Adjust
131     (S : in out Unbounded_Stack)
132   is
133   begin
134     Put_Line ("Adjusting Unbounded_Stack...");
135     Allocate_Duplicate_Data (S);
136   end Adjust;
137
138   procedure Finalize
139     (S : in out Unbounded_Stack)
140   is
141     Last : constant Positive
142           := S.Data'Last;
143   begin
144     Put_Line ("Finalizing Unbounded_Stack "
145              & "(1 .. "
146              & Last'Image
147              & ")");
148     if S.Data /= null then
149       Free (S.Data);
150     end if;
151   end Finalize;
152
153 end Unbounded_Stacks;

```

Listing 1739: show_unbounded_stack.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Unbounded_Stacks;
4
5  procedure Show_Unbounded_Stack is
6
7    package Unbounded_Integer_Stacks is new
8      Unbounded_Stacks (Element => Integer);
9    use Unbounded_Integer_Stacks;
10
11   procedure Print_Pop_Stack
12     (S : in out Unbounded_Stack;
13      Name : String)
14   is
15     V : Integer;
16   begin
17     Put_Line ("STACK: " & Name);
18     Put ("= ");
19     while not Is_Empty (S) loop
20       V := Pop (S);
21       Put (V'Image & " ");
22     end loop;
23     New_Line;
24   end Print_Pop_Stack;
25
26   Stack : Unbounded_Stack;
27   Stack_2 : Unbounded_Stack;
28   begin
29     for I in 1 .. 10 loop
30       Push (Stack, I);
31     end loop;
32

```

(continues on next page)

(continued from previous page)

```

33   Stack_2 := Stack;
34
35   for I in 11 .. 20 loop
36       Push (Stack, I);
37   end loop;
38
39   Print_Pop_Stack (Stack, "Stack");
40   Print_Pop_Stack (Stack_2, "Stack_2");
41
42 end Show_Unbounded_Stack;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
↳ Unbounded_Stacks
MD5: 22c795f2dfd2fbdf5468b54722d7126b

Runtime output

```

Initializing Unbounded_Stack (1 .. 5)
Initializing Unbounded_Stack (1 .. 5)
Increasing Unbounded_Stack (1 .. 5) to (1 .. 10)
Finalizing Unbounded_Stack (1 .. 5)
Adjusting Unbounded_Stack...
Duplicating data for new Unbounded_Stack (1 .. 10)
Increasing Unbounded_Stack (1 .. 10) to (1 .. 15)
Increasing Unbounded_Stack (1 .. 15) to (1 .. 20)
STACK: Stack
= 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
STACK: Stack_2
= 10 9 8 7 6 5 4 3 2 1
Finalizing Unbounded_Stack (1 .. 10)
Finalizing Unbounded_Stack (1 .. 20)

```

Let's first focus on the `Unbounded_Stack` type from the `Unbounded_Stacks` package. The actual stack is implemented via the array that we allocate for the `Data` component. The initial allocation takes place in the `Initialize` procedure, which is called when an object of `Unbounded_Stack` type is created. The corresponding deallocation of the stack happens in the `Finalize` procedure.

In the `Push` procedure, we check whether the stack is full or not before storing a new element into the stack. If the stack is full, we call the `Increase_Size` procedure to *increase* the size of the array. This is actually done by calling the `Reallocate_Data` procedure, which allocates a new array for the stack and copies the original data to the new array.

Also, when copying an unbounded stack object to another object of this type, a call to the `Adjust` procedure is triggered — we do this by the assignment `Stack_2 := Stack` in the `Show_Unbounded_Stack` procedure. In the `Adjust` procedure, we call the `Allocate_Duplicate_Data` procedure to allocate a new array for the stack data and copy the data from the original stack. (Internally, the `Allocate_Duplicate_Data` procedure calls the `Reallocate_Data` procedure, which we already mentioned.)

By encapsulating the access type handling in controlled types, we can ensure that the access objects are handled correctly: no incorrect pointer usage or memory leak can happen when we use this strategy.

Encapsulating file handling

Controlled types can be used to encapsulate file handling, so that files are automatically created and closed. A common use-case is when a new file is expected to be created or opened when we declare the controlled object, and closed when the controlled object gets out of scope.

A simple example is the one of a logger, which we can use to write to a logfile by simple calls to `Put_Line`:

Listing 1740: loggers.ads

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Finalization;
3
4 package Loggers is
5
6     type Logger (<>) is
7         limited private;
8
9     function Init (Filename : String)
10         return Logger;
11
12     procedure Put_Line (L : Logger;
13                       S : String);
14
15 private
16
17     type Logger is new
18         Ada.Finalization.Limited_Controlled with
19         record
20             Logfile : File_Type;
21         end record;
22
23     procedure Finalize
24         (L : in out Logger);
25
26 end Loggers;
```

Listing 1741: loggers.adb

```
1 package body Loggers is
2
3     --
4     --  SUBPROGRAMS
5     --
6
7     function Init (Filename : String)
8         return Logger is
9     begin
10         return L : Logger do
11             Create (L.Logfile, Out_File, Filename);
12         end return;
13     end Init;
14
15     procedure Put_Line (L : Logger;
16                       S : String) is
17     begin
18         Put_Line ("Logger: Put_Line");
19         Put_Line (L.Logfile, S);
20     end Put_Line;
21
```

(continues on next page)

(continued from previous page)

```

22  --
23  --  PRIVATE SUBPROGRAMS
24  --
25
26  procedure Finalize
27  (L : in out Logger) is
28  begin
29      Put_Line ("Finalizing Logger...");
30      if Is_Open (L.Logfile) then
31          Close (L.Logfile);
32      end if;
33  end Finalize;
34
35  end Loggers;

```

Listing 1742: some_processing.adb

```

1  with Loggers; use Loggers;
2
3  procedure Some_Processing (Log : Logger) is
4  begin
5      Put_Line (Log, "Some processing...");
6  end Some_Processing;

```

Listing 1743: show_logger.adb

```

1  with Loggers;      use Loggers;
2  with Some_Processing;
3
4  procedure Show_Logger is
5      Log : constant Logger := Init ("report.log");
6  begin
7      Put_Line (Log, "Some info...");
8      Some_Processing (Log);
9  end Show_Logger;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
↳ Logger
MD5: 0ac4b5dff9ded8b64cb2b1f001e763fa

Some info...
Some processing...

Runtime output

Logger: Put_Line
Logger: Put_Line
Finalizing Logger...

The Logger type from the Loggers package has two subprograms:

- Init, which creates a logger object and creates a logfile *in the background*, and
- Put_Line, which writes a message to the logfile.

Note that we use the (<>) in the declaration of the Logger type to ensure that clients call the Init function. This allows us to specify the location of the logfile (as the Filename parameter).

Also, we can pass the logger to other subprograms and use it there. In this example, we

pass the logger to the `Some_Processing` procedure and there, we the call `Put_Line` using the logger object.

Finally, as soon as the logger goes out of scope, the log is automatically closed via the call to `Finalize`.

For further reading...

Instead of enforcing a call to `Init`, we could have overridden the `Initialize` procedure and opened the logfile there. This approach, however, would have prevented the client from specifying the location of the logfile in a simple way. Specifying the filename as a type discriminant wouldn't work because we cannot use a string as a discriminant — as we mentioned *in a previous chapter* (page 467), we cannot use indefinite subtypes as discriminants.

If we had preferred this approach, we could generate a random name for the file in the `Initialize` procedure and store the file itself in a temporary directory indicated by the operating system. Alternatively, we could use the access to a string as a discriminant:

Listing 1744: `loggers.ads`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Finalization;
3
4 package Loggers is
5
6     type Logger (Filename : access String) is
7         limited private;
8
9     procedure Put_Line (L : Logger;
10                        S : String);
11
12 private
13
14     type Logger (Filename : access String) is new
15         Ada.Finalization.Limited_Controlled with
16         record
17             Logfile : File_Type;
18         end record;
19
20     procedure Initialize
21         (L : in out Logger);
22
23     procedure Finalize
24         (L : in out Logger);
25
26 end Loggers;
```

Listing 1745: loggers.adb

```

1 package body Loggers is
2
3   --
4   --  SUBPROGRAMS
5   --
6
7   procedure Put_Line (L : Logger;
8                      S : String) is
9   begin
10      Put_Line ("Logger: Put_Line");
11      Put_Line (L.Logfile, S);
12   end Put_Line;
13
14   --
15   --  PRIVATE SUBPROGRAMS
16   --
17
18   procedure Initialize
19     (L : in out Logger) is
20   begin
21      Create (L.Logfile,
22             Out_File,
23             L.Filename.all);
24   end Initialize;
25
26   procedure Finalize
27     (L : in out Logger) is
28   begin
29      Put_Line ("Finalizing Logger...");
30      if Is_Open (L.Logfile) then
31         Close (L.Logfile);
32      end if;
33   end Finalize;
34
35 end Loggers;

```

Listing 1746: show_logger.adb

```

1 with Loggers;      use Loggers;
2 with Some_Processing;
3
4 procedure Show_Logger is
5   Name : aliased String := "report.log";
6   Log : Logger (Name'Access);
7 begin
8   Put_Line (Log, "Some info...");
9   Some_Processing (Log);
10 end Show_Logger;

```

Code block metadata

Project: Courses.Advanced_Ada.Resource_Management.Controlled_Types.Applications.
 ↪Logger
 MD5: d60ffbfafd26d3d70a3d7807487dd95ab

Some info...
 Some processing...

Runtime output

Logger: Put_Line
 Logger: Put_Line
 Finalizing Logger...

This approach works, but requires us to declare an aliased string (Name), which we can give access to in the declaration of the Log object.

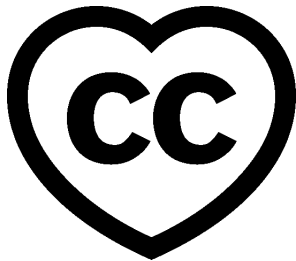
By encapsulating the file handling in controlled types, we ensure that files are properly opened when we want to use them, and that the files are closed when they're not going to be used anymore.

Part III

Ada In Practice

Copyright © 2025, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)³²⁸



This course explores how to use the Ada language in real-world scenarios that extend beyond the complexity of lab exercises. The resulting solutions to these scenarios are idioms and techniques, some common to various programming languages and some specific to Ada. Multiple preliminary solutions to a given scenario (when they exist) are presented and analyzed for strengths and weaknesses, leading to the final preferred solution and its analysis. As a result, readers — especially those relatively new to the language — can learn how experienced Ada developers apply the language in actual practice. Prior knowledge of Ada is required, although explanations of the underlying semantics are provided when appropriate.

This document was written by Patrick Rogers and reviewed by Gustavo A. Hoffmann, Steven Baird, Richard Kenner, Robert A. Duff, and Tucker Taft.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](#)³²⁹. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;

³²⁸ [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

³²⁹ https://learn.adacore.com/zip/learning-ada_code.zip

INTRODUCTION

This course describes how to implement selected programming idioms in the Ada language. What is an idiom? Some would say that an idiom is a workaround for an expressive deficiency in a programming language. That is not what we mean.

What we have in mind are answers to the question "In this situation, what is the most elegant implementation approach?". Elegant software is comprehensible, efficient, concise, reliable, and, as a result, maintainable, so elegance is an economically and technically desirable characteristic.

Design patterns³³⁶ are intended to answer that question, and indeed some would equate idioms with design patterns. But what we have in mind is more general in scope.

For example, [Reference Counting](#)³³⁰ is a well-known approach to tracking and managing the storage for objects and is conceptually independent of the programming language. However, reference counting is not a design pattern.

Likewise, [Resource Acquisition Is Allocation \(RAII\)](#)³³¹, [type punning](#)³³², [interface inheritance](#)³³³, and [implementation inheritance](#)³³⁴ are not design patterns.

Those are the kinds of situations and implementations we focus upon.

That said, we may refer to a design pattern to illustrate an idiom's purpose and/or implementation. For example, in the idiom for controlling object creation and initialization, the implementation approach happens to be the same as for expressing a Singleton³³⁶.

In addition to language-independent situations, we also include implementations for situations specific to the Ada language. These idioms are *best practices* in situations that arise given the extensive capabilities of the language.

For example, Ada directly supports tasks (threads) via a dedicated construct consisting of local objects and a sequence of statements. Tasks can also be defined as types, and then used to define components for other composite types. As a result, there is an idiom showing how to associate a task type with an enclosing composite type so that the task components have visibility to the enclosing object's other components.

In all the idioms we want to apply the fundamental principles of software engineering, especially those of abstraction and information hiding. Therefore, we include an idiom for expressing abstractions as types, with compile-time visibility control over the representation. These are the well-known Abstract Data Types, something the Ada language directly supports but using *building blocks* instead of a single construct. For that same reason we include another idiom for defining abstractions that manage global data (Abstract Data Machines). Most of the idioms' implementations will be defined using these abstraction techniques as their starting point.

³³⁶ E. Gamma, R. Helm, and others. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley Publishing Company, 1995.

³³⁰ https://en.wikipedia.org/wiki/Reference_counting

³³¹ https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

³³² https://en.wikipedia.org/wiki/Type_punning

³³³ <https://en.wikipedia.org/wiki/Subtyping>

³³⁴ https://en.wikipedia.org/wiki/Implementation_inheritance

29.1 Assumptions

We assume the reader knows Ada to some degree, including some advanced topics. For those lacking significant familiarity, we hope these implementations will at least give a sense for how to apply the language. We direct such readers to the [online Learn courses dedicated to the Ada language itself](#) (page 277).

29.2 Definitions

For the sake of avoiding duplication in the idiom entries, the following terms are defined here. Note that the Ada Language Manual includes a glossary in [Section 1.3](#)³³⁵ (located in Annex N prior to Ada 2022). Some of the following expand on the definitions found there.

29.2.1 Suppliers and Clients

Suppliers are software units that provide programming entities to other software units, the users. These users are the *clients* of the supplied units. The concept is simple and intuitive, but by defining these terms we can convey these roles quickly in the idioms' discussions.

For example, a unit that defines a type and associated operations would be a supplier. Client units could use that type to declare objects, and/or apply the operations to such objects. The language-defined package `Ada.Text_IO` is an example of a supplier. Similarly, the unit that defines a library, such as a math library, is a supplier. Callers to the math library routines are the clients. The generic package `Ada.Numerics.Generic_Complex_Elementary_Functions`, once instantiated, would be an example supplier. (Arguably, the generic package itself would be a supplier to the client that instantiates it, but instantiation is the only possibility in that narrow case. Only the routines in the instances can be called.)

Bertrand Meyer's book on OOP³³⁷ limits these terms specifically to the case of a type used in an object declaration. Our definitions cover that case but others as well.

Units can be both suppliers and clients, because a given supplier's facility, i.e., the interface and/or implementation, may be built upon the facilities defined by other suppliers.

29.2.2 Compile-time Visibility

In the definitions of supplier and client above, we gave an example in which a supplier's type was used by clients to declare objects of the type. For the client to legally do so — that is, for the compiler to accept this usage and process the code — the use of the supplier's type has to satisfy the scope and visibility rules of the programming language.

Good implementations harness these visibility rules to adhere to the software engineering principles of information hiding and abstraction, both of which require that nothing of the implementation be made visible to clients unless necessary. Compiler enforcement ensures rigorous adherence to those principles.

Therefore, modern languages provide some way to express this control. For example, in Ada, a package can have both a *public* part and a *private* part. Clients have no compile-time visibility to the private part, nor to the package body, as both parts contain implementation artifacts. In class-oriented languages, parts of the class can be marked as *public*, *private*, and *protected* (the details depend on the specific language).

The idioms [Abstract Data Types](#) (page 1187) and [Abstract Data Machines](#) (page 1193) are prime examples used throughout the other idioms.

³³⁵ <http://www.ada-auth.org/standards/22rm/html/RM-1-3.html>

³³⁷ B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

The idioms explored in *Fundamental Packages* (page 1183) are largely variations on expressing this control in Ada.

More details on the topic are provided in those idioms.

29.2.3 Views

In Ada, a *view* of an entity defines what the developer can legally do with that entity. For example, the declaration of an object defines a view of that object. The operations allowed by that view are determined by the type used to declare the object: a signed integer type would allow signed integer numeric operations, but not, say, bit-level operations, nor array indexing, and so on. Furthermore, the view includes whether the object is a constant.

An entity can have more than one view, depending on where in the text of the source code a view of that entity is considered. For example, let's say that the integer object introduced above is in fact a variable. Within the scope of that variable, we can refer to it by that name and update the value using assignment statements. However, if we pass that variable as the argument to a procedure call, within that subprogram (for that call) the view specifies a different name for the argument, i.e., the formal parameter name. Moreover, if that formal parameter is a mode-in parameter, within that procedure body the view of the actual parameter is as if it were a constant rather than a variable. No assignments via the formal parameter name are allowed because the view at that point in the text — within that procedure body — doesn't allow them, unlike the view available at the point of the call.

As another example, consider a tagged type named *Parent*, and a type derived from it via type extension, named *Child*. It is common for a derived type to have either additional components, or additional operations, or both. For a given object of the *Child* type, the view via type *Child* allows the developer to refer to the extended components and/or operations. But we can convert the *Child* object to a value of the *Parent* type using what is known as a *view conversion*. With that *Parent* view of the *Child* object, we can only refer to those components and operations defined for the *Parent* type. The compiler enforces this temporary view.

For further details about view conversions, please refer to that *specific section of the Advanced Ada course* (page 328).

Views are a fundamental concept in Ada. Understanding them will greatly facilitate understanding the rules of the language in general.

29.2.4 Partial and Full Views

Like objects, types also can have more than one view, again determined by the place in the program text that a view is considered. These views can be used to apply information hiding and abstraction.

The declaration of a private type defines a *partial view* of a type that reveals only some of its properties: the type name, primarily, but in particular not the type's representation. For example:

```
type Rotary_Encoder is private;
```

Private type declarations must occur in the *public part* of a package declaration. Anything declared there is compile-time visible to clients of the package so the type's name is visible, and potentially some other properties as well. Clients can therefore declare objects of the type name, for example, but must adhere to their partial view's effect on what is compile-time visible.

The private type's full representation must be specified within the *private part* of that same package declaration. For example:


```
type Rotary_Encoder is record ... end record;
```

Therefore, within that package private part and within the package body the *full view* is available because full representation information is compile-time visible in those regions. (Parts of child units have the full view as well.) This view is necessary in those two regions of the package because the representation details are required in order to implement the corresponding operations, among other possibilities.

Because the clients only have the partial view they do not have compile-time visibility to the type's internal representation. Consequently, the compiler will not allow representation-specific references or operations in client code. The resulting benefit is that clients are independent of the type's representation and, therefore, it can be changed without requiring coding changes in the clients. Clients need only be recompiled in that case.

This application of information hiding has real-world cost benefits because changing client code can be prohibitively expensive. That's one reason why the maintenance phase of a project is by far the most expensive phase. Another reason is that *maintenance* is often a euphemism for new development. Either way, change is involved.

As a result, when defining types, developers should use private types by default, only avoiding them when they are not appropriate. Not using them should be an explicit design choice, a line item in code reviews. Not defining a major abstraction as a private type should be suspect, just as using a **struct** rather than a `class` in C++ should be suspect in that case. (In C++ anything a **struct** contains is compile-time visible to clients by default.)

For further details about type views, please refer to that [specific section of the Advanced Ada course](#) (page 312).

ESSENTIAL DESIGN IDIOMS FOR PACKAGES

30.1 Motivation

Packages, especially library packages, are modules, and as such are the fundamental building blocks of Ada programs. There is no language-prescribed way to use packages when designing an application, the language just specifies what is legal. However, some legal approaches are more advisable than others.

Specifically, packages should exhibit high cohesion and loose coupling³³⁸. Cohesion is the degree to which the declarations within a module are related to one another, in the context of the problem being solved. Unrelated entities should not be declared in the same module. This allows the reader to focus on one primary concept, which should be the subject of the package. Coupling is the degree to which a module depends upon other modules. Loose coupling enhances comprehension and maintenance because it allows readers and future developers to examine and modify the module in relative isolation. Coupling and cohesion are interrelated: higher cohesion tends to result in less coupling.

30.2 Implementation(s)

Three idioms for packages were envisioned when the language was first designed. They were introduced and described in detail in the Rationale document for the initial language design³³⁹ and were further developed in Grady Booch's book *Software Engineering with Ada*³⁴⁰, a foundational work on design with the (sequential part of the) language. Booch added a fourth idiom, the Abstract Data Machine, to the three described by the Rationale. These four idioms have proven themselves capable of producing packages that exhibit high cohesion and loose coupling, resulting in more comprehensible and maintainable source code.

These idioms pre-date later package facilities, such as private packages and hierarchical packages. We describe idioms for those kinds of packages separately.

Two of the simpler idioms are described here. The other two, that are more commonly used, are described in two separate, dedicated entries within this document.

Generic packages are not actually packages, but their instantiations are, so these design idioms apply to generic packages as well.

Because these are idioms for modules, we differentiate them by what the package declarations will contain. But as you will see, what they can contain is a reflection of the degree of information hiding involved.

³³⁸ E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.

³³⁹ J. Ichbiah, J. Barnes, and others. *Rationale for the Design of the Ada Programming Language*. 1986.

³⁴⁰ G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.

30.2.1 Named Collection of Declarations

In the first idiom, the package declaration can contain other declarations only for the following:

- Objects (constants and variables)
- Types
- Exceptions

The idea is to factor out common content required by multiple clients. Declaring common content in one place and letting clients reference the one unit makes the most sense.

For example, the following package declares several physical constants used in a high-fidelity aircraft simulator. These constants are utilized throughout the simulator code, so they are declared in one place and then referenced as needed:

```
package Physical_Constants is
  Polar_Radius    : constant := 20_856_010.51; -- feet
  Equatorial_Radius : constant := 20_926_469.20; -- feet
  Earth_Diameter : constant :=
    2.0 * ((Polar_Radius + Equatorial_Radius)/2.0);
  Gravity : constant := 32.1740_4855_6430_4; -- feet/second**2
  Sea_Level_Air_Density : constant := 0.002378; -- slugs/foot**3
  Altitude_Of_Tropopause : constant := 36089.0; -- feet
  Tropopause_Temperature : constant := -56.5; -- degrees-C
end Physical_Constants;
```

No information hiding is occurring when using this idiom.

Pros

Packages designed with this idiom will have high cohesion and low coupling.

The idiom also enhances maintainability because changes to the values, if necessary, need only be made in one place, although in this particular example, we would hope that no such changes will be made.

Cons

When a library package contains variable declarations, these variables comprise global data. In this sense, *global* means potential visibility to multiple clients. Global data should be avoided by default, because the effects of changes are potentially pervasive, throughout the entire set of clients that have visibility to it. In effect the developer must understand everything before changing anything. The introduction of new bugs is a common result. But if, for some compelling reason, the design really called for global data, this idiom provides the way to declare it. Note also that global *constants* are less problematic than variables because they can't be changed.

30.2.2 Groups of Related Program Units

In this idiom, the package can contain all of the declarations allowed by the first idiom, but also contains declarations for operations. These are usually subprograms but other kinds of declarations are also allowed such as protected types and objects. Hence these packages can contain:

- Objects (constants and variables)
- Types
- Exceptions
- Operations

Our intent is that the types declared in the package are used by the operations declared in the package, typically in their formal parameters and/or function return types. In this idiom, however, the types are not private.

For example:

```
package Linear_Algebra is
  type Vector is array (Positive range <>) of Real;
  type Matrix is array (Positive range <>, Positive range <>) of Real;
  function "+" (Left, Right : Vector) return Vector;
  function "*" (Left, Right : Vector) return Matrix;
  -- ...
end Linear_Algebra;
```

In this example, `Vector` and `Matrix` are the types under consideration. The type `Real` might be declared here too, but it might be better declared in a *Named Collection of Declarations* (page 1184) package referenced in a `with_clause`. In any case, this package declares types and subprograms that manipulate values of those types.

One might also declare variables in the package, but those should not be the central purpose of the package. For example, perhaps we want to have a variable whose value is used as the default for some formal parameters. Clients can change the default for subsequent calls by first assigning a different value to the variable, unlike a hardcoded literal chosen by the developer. It would look like this:

```
Default_Debounce_Time : Time_Span := Milliseconds (75);
-- The default amount of time used to debounce an input pin.
-- This value is tunable.

procedure Await_Active
  (This : Discrete_Input;
   Debounce_Time : Time_Span := Default_Debounce_Time);
```

With this idiom, information hiding applies to the implementation of the visible subprograms in the package body as well as any internal entities declared in the body and used in implementing the visible subprograms.

As mentioned, these idioms apply to generic packages as well. For example, a more realistic approach would be to make type `Real` be a generic formal type:

```
generic
  type Real is digits <>;
package Linear_Algebra is
  type Vector is array (Positive range <>) of Real;
  type Matrix is array (Positive range <>, Positive range <>) of Real;
  function "+" (Left, Right : Vector) return Vector;
  function "*" (Left, Right : Vector) return Matrix;
  -- ...
end Linear_Algebra;
```

Pros

The types and the associated operations are grouped together and are hence highly cohesive. Such packages usually can be loosely coupled as well.

Clients have all the language-defined operations available that the type representations provide. In the case of `Vector` and `Matrix`, clients have compile-time visibility to the fact they are array types. Therefore, clients can manipulate `Vector` and `Matrix` values as arrays: for example, they can create values via aggregates and use array indexing to access specific components.

Cons

Clients can write code that depends on the type's representation, and can be relied upon to do so. Consequently, a change in the representation will potentially require redeveloping the client code, which could be extensive and expensive. That is a serious disadvantage.

However, compile-time visibility to the type representations may be necessary to meet client expectations. For example, engineers expect to use indexing with vectors and matrices. As of Ada 2012, developers can specify the meaning of array indexing but the approach is fairly heavy.

30.3 Notes

1. The rules for what these idiomatic packages contain are not meant to be iron-clad; hybrids are possible but should be considered initially suspect and reviewed accordingly.

ABSTRACT DATA TYPES

31.1 Motivation

In the *Groups of Related Program Units* (page 1184) idiom, client compile-time visibility to the type's representation is both an advantage and a disadvantage. Visibility to the representation makes available the expressiveness of low-level syntax, such as array indexing and aggregates, but in so doing allows client source code to be dependent on the representation. In many cases, the resulting economic and engineering disadvantages of visibility on the representation will outweigh the expressiveness advantages.

For the sake of illustration, let's create a *stack* type that can contain values of type **Integer**. (We use type **Integer** purely for the sake of convenience.) Let's also say that any given Stack object can contain at most a fixed number of values, and arbitrarily pick 100 for that upper bound. The likely representation for the Stack type will require both an array for the contained values and a *stack pointer* indicating the *top* of the stack. Hence this will be a composite type, probably a record type. If we use the *Groups of Related Program Units* (page 1184) idiom the code might look like this:

```
package Integer_Stacks is
  Capacity : constant := 100;
  type Content is array (1 .. Capacity) of Integer;
  type Stack is record
    Values : Content;
    Top    : Integer range 0 .. Capacity := 0;
  end record;
  procedure Push (This : in out Stack; Item : in Integer);
  procedure Pop  (This : in out Stack; Item : out Integer);
  function Empty (This : Stack) return Boolean;
end Integer_Stacks;
```

With this design the compiler will allow client code to directly read and update the two components within any Stack object. For example, given some Stack variable named X, the client can read the value of X.Top, say to determine if X is empty. But by the same token, the client code could change X.Top to some arbitrary value unrelated to the logical top of the stack, completely violating stack semantics.

As a result, where would one look in the source code to find a bug in the handling of some Stack object? It could be literally anywhere in all the client code that uses package Integer_Stacks.

Similarly, changes to the internal representation of a type may become necessary as new requirements are identified. At best, the client code will now fail to compile, making identification of the problem areas simple. At worst, the client code will remain legal but no longer functional. Perhaps an additional component was added that the original components now rely upon, or the original components are used in new ways. Conceivably every client use of Integer_Stacks might need to be changed. Once we find them all we'll have to rewrite them to address the changes in the representation. That's potentially very expensive, perhaps prohibitively so. Worse, our *fixes* will likely introduce new bugs.

These disadvantages argue for an alternative. That is the purpose of this next idiom, known as the Abstract Data Type (ADT)^{341, 342}.

31.2 Implementation(s)

Abstraction is one of the central principles of software engineering because it is one of the primary ways that humans manage complexity. The idea is to focus on the essentials, in effect the *what*, while ignoring all the inessential implementation details, i.e., the *how*. For example, when we drive a car and want to stop, we press the brake pedal. We don't also think about how the pedal makes the car stop, just that it does so. That's an example of abstraction. In the same way, we know that pressing the accelerator pedal increases the speed of the car, that rotating the steering wheel changes the direction of travel, and so on. If to control the car we had to think about how each part actually works — the brake cylinder and brake pads, the fuel injectors, the spark plugs, the steering shaft, the tie rods, and everything else — we'd certainly crash.

We use abstraction in programming for the same reason. In higher-level languages, an array is an abstraction for the combination of a base address and offset. A file system is composed of a number of layered abstractions, including files (at the top), then tracks, then sectors, then blocks, and ultimately down to individual bytes. A data structure, such as a stack, a queue, or a linked list, is an example of an abstraction, as is a valve, an air-lock, and an engine when represented in software. Even procedures and functions are abstractions for lower-level operations. Decomposing via abstractions allows us to manage complexity because at any given layer we can focus on *what* is being done, rather than how.

Therefore, an abstract data type is a type that is abstract in the sense that³⁴²:

- It is a higher level of abstraction than the built-in programming language types.
- It is functionally characterized entirely by the operations defined by the ADT itself, along with the common basic operations such as assignment, object declarations, parameter passing, and so on. In particular, clients are not allowed to perform operations that are determined by the type's internal representation. Ideally, this protection is enforced by tools.

The ADT may also be abstract in the sense of object-oriented programming but that is an unrelated issue.

In Ada we use *private types* to define abstract data types because private types make the type's name, but not its representation, visible to clients. These types are composed using syntactical building blocks: a package declaration, separated into two parts, containing a type declared in two parts, and containing declarations for subprograms to manipulate objects of the type via parameters. The compiler uses the building-blocks' compile-time visibility rules to enforce the protections against representation-based operations. (We assume the reader is familiar with private types, but this is such an important, central facility in Ada that we will explain them in some detail anyway.)

Therefore, an ADT package declaration may contain any of the following:

- Constants (but probably not variables)
- A private type
- Ancillary Types
- Exceptions
- Operations

³⁴¹ G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.

³⁴² B. Liskov and S. Zilles. *Programming with Abstract Data Types*. ACM SIGPLAN symposium on Very high level languages, 1974.

If possible, you should declare at most one private type per ADT package. This keeps things simple and follows the "cohesive" principle. (Note that the *limited-with* construct directly facilitates declaring mutually-dependent private types that are each declared in their own dedicated packages). However, it's not unreasonable to declare more than one private type in the same package, especially if one of the types is clearly the primary type and the other private type is related to the first. For example, in defining an ADT for a maze, we could declare a private type named `Maze` to be the primary abstraction. But mazes have positions within them, and as clients have no business knowing how positions are represented, both `Maze` and `Position` could reasonably be declared as private types in the same package.

You may use any form of private type with this idiom: basic private types, tagged/abstract/limited private types, private type extensions, and so forth. What's important is that the representation occurs in the private part so that it's not compile-time visible to clients.

The abstraction's operations consist of subprograms that each have one or more formal parameters of the type. Clients will declare objects of the type and pass these objects as formal parameters to manipulate those objects.

The operations are known as *primitive operations* because they have the compile-time visibility to the private type's representation necessary to implement the required behavior.

Clients can create their own operations by calling the type's primitive operations, but client's can't compile any operation that manipulates the internal representation.

Consider the following revision to the package `Integer_Stacks`, now as an ADT:

```
package Integer_Stacks is
  type Stack is private;
  procedure Push (This : in out Stack; Item : in Integer);
  procedure Pop  (This : in out Stack; Item : out Integer);
  function Empty (This : Stack) return Boolean;
  Capacity : constant := 100;
private
  type Content is array (1 .. Capacity) of Integer;
  type Stack is record
    Values : Content;
    Top    : Integer range 0 .. Capacity := 0;
  end record;
end Integer_Stacks;
```

The package declaration now includes the **private** reserved word, about half-way down by itself in the example above, thus dividing the package declaration into the *public part* and the *private part*. The compiler only allows clients compile-time visibility to the package public part. No client code that references anything in the private part will compile successfully.

The declaration for the type `Stack` now has two pieces, one in the package visible part and one in the package private part. The visible piece introduces the type name and ends with the keyword **private** to indicate that its representation is not provided to clients.

Client code can use the type name to declare objects because the name is visible. Likewise, clients can declare their own subprograms with parameters of type `Stack`, or use type `Stack` as the component type in a composite type declaration. Clients can use a private type in any way that's consistent with the rest of the visible type declaration, except they can't see anything representation-dependent.

The full type definition is in the package private part. Therefore, for any given object of the type, the representation details — the two record components in this example — can't be referenced in client code. Clients must instead only use the operations defined by the package, passing the client objects as the actual parameters. Only the bodies of these operations have compile-time visibility to the representation of the `Stack` parameters, so only they can implement the functionality for those parameters.

Because package-defined subprograms are the only code that can access the internals of objects of the type, the designer's intended abstract operations are strictly enforced. They are the only manipulations that a client can perform. As we mentioned, basic operations such as assignment are allowed, unless the ADT is *limited* as well as private, but these basic operations do not violate the abstraction.

You may, of course, also require other ancillary type declarations in the package, either for the implementation or as types for additional parameters for the visible operations. The array type `Content` is an example of the former case. When it is strictly an implementation artifact, as in this case, it should be in the private part so that it's hidden from clients.

The ADT idiom extends the information hiding applied by the *Groups of Related Program Units* (page 1184) idiom to include the type's representation.

The compile-time lack of visibility to the representation means that clients no longer have a way to construct ADT values from the constituent parts. For example, record aggregates are no longer possible for clients using the Stack ADT. Likewise, clients no longer have a way to read the individual constituent components. (Whether doing so is appropriate will be addressed below.) Therefore, an ADT package may include *constructor* and *selector/accessor* subprograms. (The term *constructor* is only conceptually related to the same term in some other languages, such as C++.)

For an example of an abstraction that includes constructors and selectors, imagine there is no language-defined Complex number type. We could use the following ADT approach:

```
package Complex_Numbers is
  type Complex_Number is private;
  -- function operating on Complex_Number, eg "+" ...
  -- constructors and selectors/accessors
  function Make (Real_Part, Imaginary_Part : Float) return Complex_Number;
  function Real_Part (This : Complex_Number) return Float;
  function Imaginary_Part (This : Complex_Number) return Float;
private
  type Complex_Number is record
    Real_Part : Float;
    Imaginary_Part : Float;
  end record;
end Complex_Numbers;
```

In the above, the function `Make` is a constructor that replaces the use of aggregates for constructing `Complex_Number` values. Callers pass two floating-point values to be assigned to the components of the resulting record type. In the Stack ADT, a constructor for Stack objects wasn't required because any stack has a known initial state, i.e., empty, and the component default initialization is sufficient to achieve that state. Complex numbers don't have any predeterminable state so the constructor is required.

Likewise, functions `Real_Part` and `Imaginary_Part` are selector/accessor functions that return the corresponding individual component values of an argument of type `Complex_Number`. They are needed because the mathematical definition of complex numbers has those two parts, so clients can reasonably expect to be able to get such values from a given object. (The function names need not be distinct from the component names, but can be if desired.)

However, by default, selector/accessor functions are not included in the ADT idiom, and especially not for every component of the representation. There are no *getter* operations if you are familiar with that term.

There may be cases when what looks like an accessor function is provided, when in fact the function computes the return value. Similarly, there may be functions that simply return the value of a component but are part of the abstraction and happen to be implementable by returning the value of a component. For example, a real stack's ADT package would include a function indicating the extent of the object — that is, the number of values currently contained. In our example implementation the `Top` component happens to indicate that

value, in addition to indicating the current top of the stack. The body of the Extent function can then be as follows:

```
function Extent (This : Stack) return Natural is (This.Top);
```

But a different representation might not have a Top component, in which case the function would be implemented in some other way. (For example, we could have declared a subtype of **Natural**, using Capacity as the upper bound, for the function result type.)

You should not include true *getter* functions that do not meet an abstraction-defined requirement and exist purely to provide client access to the otherwise hidden representation components included. Their usage makes the client code dependent on the representation, just as if the client had direct access. For the same reason, by default there are no *setter* procedures for the representation components. Both kinds of operations should be considered highly suspect. There's no point in hiding the representation if these operations will make it available to clients, albeit indirectly.

31.3 Pros

The advantages of an ADT are due to the strong interface presented, with guaranteed enforcement by the compiler rather than by reliance on clients' good behavior. The ADT designer can rely on client adherence to the intended abstraction because client code that violates the designer's abstraction by directly manipulating the internals of the type will not compile; clients must call the designer's operations to manipulate the objects.

A package defining a strong interface will exhibit high cohesion, thereby aiding comprehension and consequently easing both development and maintenance.

An ADT enhances maintainability because a bug in the ADT implementation must be in the package that defines the ADT itself. The rest of the application need not be explored because nothing elsewhere that accessed the representation would compile. (We ignore child packages for the time-being.) The maintenance phase is the most expensive of the project phases for correcting errors, so this is a significant benefit.

Although changes to the internal representation of an ADT may become necessary, the scope of those changes is limited to the ADT package declaration and body because legal client code cannot depend on the representation of a private type. Consequently, changes to the type's representation can only require recompilation (and hence relinking) of client code, but not rewriting.

A change in representation may have non-functional considerations that prompt a change in client usage, such as performance changes, but it will not be a matter of the legality of the client code. Illegal client usage of an ADT wouldn't have compiled successfully in the first place.

The private type is the fundamental approach to creating abstractions in Ada, just as the use of the *public*, *private*, and *protected* parts of classes is fundamental to creating abstractions in class-oriented languages. Not every type can be private, as illustrated by the client expectation for array indexing in Ada prior to Ada 2012. Not every type should be private, for example those that are explicitly numeric. But the ADT should be the default design idiom expression.

31.4 Cons

There is more source code text required in an ADT package compared to the idiom in which the representation is not hidden (the *Groups of Related Program Units* (page 1184)). The bulk of the additional text is due to the functions and procedures required to provide the capabilities that the low-level representation-based syntax might have provided, i.e., the *constructor* and *selector/accessor* functions. We say *might have provided* because these

additional operations are by no means necessarily included. In general, the additional text required for private types is worth the protections afforded.

31.5 Relationship With Other Idioms

The package-oriented idioms described here and *previously* (page 1183) are the foundational program composition idioms because packages are the primary structuring unit in Ada. That is especially true of the *Abstract Data Type* (page 1187) idiom, which is the primary type specification facility in Ada. We will describe additional package-oriented idioms, especially regarding hierarchical packages, but those kinds of packages are optional.

The basic package is not optional in Ada for a program of any significant size or complexity. (One could have a program consisting entirely of the main program, but either that program is relatively simple and small or it is badly structured.) As a consequence, other idioms will exist within packages designed using one of these idioms or some other package idiom.

31.6 Notes

1. With the package idioms that declare one or more types, especially the ADT idiom, the principle of Separation of Concerns dictates that objects of the type used by clients be declared by clients in client units, not in the same package that declares the type or types.
2. The Ada Rationale document did not introduce the concept of Abstract Data Types. The ADT concept had already been introduced and recognized as effective when the first version of Ada was being designed^{Page 1188, 342}. The Ada language requirements document, *Steelman*³⁴³, uses the term "Encapsulated Definitions" and describes the information hiding to be provided. Steelman does not specify the implementation syntax because requirements documents do not include such directives. The language designers implemented those requirements via package private parts and private types.
3. The ADT is the conceptual foundation for the *class* construct's visibility control in some class-oriented languages.

³⁴³ HOLWG. *Department of Defense Requirements for High Order Computer Programming Language "STEELMAN"*. Department of Defense, 1978.

ABSTRACT DATA MACHINES

32.1 Motivation

In some systems, only one logical "instance" of an abstraction should exist in the software. This requirement may stem from the functionality involved. For example, a subsystem-level software logging facility should be unique at that level. Likewise, the function of a hardware device may be such that only one instance should exist in both the system and the software. A security device that validates users would be an example. Another reason can be simple physical reality. There might be only one on-board or on-chip device of some sort. Execution on that board or chip entails there being only one such device present.

How can the software representing the abstraction best implement this requirement?

The Abstract Data Type (ADT) *Abstract Data Type* (page 1187) idiom is the primary abstraction definition facility in Ada. Given an ADT that provides the required facility you could simply declare a single object of the type. But how could you ensure that some other client, perhaps in the future, doesn't declare another object of the type, either accidentally or maliciously?

As a general statement about program design, if there is something that must not be allowed, the ideal approach is to use the language rules to make it impossible. That's far better than debugging. For example, we don't want clients to have compile-time access to internal representation artifacts, so we leverage the language visibility rules to make such access illegal. The compiler will then reject undesired references, rigorously.

The occasional need to control object creation is well-known, so much so that there is a design pattern for creating an ADT in which only one instance can ever exist. Known as the "singleton" pattern, the given programming language's rules are applied such that only the ADT implementation can create objects of the type. Clients cannot do so. The implementation only creates one such object, so multiple object declarations are precluded.

Singletons can be expressed easily in Ada *Controlling Object Initialization and Creation* (page 1211) but there is an alternative in this specific situation.

This idiom entry describes the alternative, known as the Abstract Data Machine (ADM). The Abstract Data Machine was introduced by Grady Booch³⁴⁴ as the Abstract State Machine, but that name, though appropriate, encompasses more in computer science than we intend to evoke.

32.2 Implementation(s)

The ADM is similar to the ADT idiom in that it presents an abstraction that doesn't already exist in the programming language. Furthermore, like the ADT, operations are provided to clients to manipulate the abstraction state, which is not otherwise compile-time visible to client code.

³⁴⁴ G. Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, 1983.

Unlike the ADT, however, the ADM does not define the abstraction as a type. To understand this difference, first recall that type declarations are descriptions for objects that will contain data (the state). For example, our earlier Stack ADT was represented as a record containing two components: an array to hold the values logically contained by the Stack and an integer indicating the logical top of that array (the stack pointer). No data actually exists, i.e., is allocated storage, until objects of the type are declared. Clients can declare as many objects of type Stack as they require and each object has a distinct, independent copy of the data.

Continuing the Stack example, clients could choose to declare only one object of the Stack type, in which case only one instance of the data described by the Stack type will exist:

```
Integer_Stack : Stack;
```

But, other than convenience, there is no *functional* difference from the client declaring individual variables of the representational component types directly, one for the array and one for the stack pointer:

```
Capacity : constant := 100;
type Content is array (1 .. Capacity) of Integer;
Values : Content;
Top : Integer range 0 .. Capacity := 0;
```

or even this, using an anonymously-typed array:

```
Capacity : constant := 100;
Values : array (1 .. Capacity) of Integer;
Top : Integer range 0 .. Capacity := 0;
```

If there is to be only one logical stack, these two variables will suffice.

That's what the ADM does. The state variables are declared directly within a package, rather than as components of a type. In that way the package, usually a library package, declares the necessary state for a single abstraction instance. But, as an abstraction, those data declarations must not be compile-time visible to clients. Therefore, the state is declared in either the package private part or the package body. Doing so requires that visible operations be made available to clients, as with the ADT. Hence the combination of a package, the encapsulated variables, and the operations is the one instance of the abstraction. That combination is the fundamental concept for the ADM idiom.

Therefore, the package declaration's visible section contains only the following:

- Constants (but almost certainly not variables)
- Ancillary Types
- Exceptions
- Operations

The package declaration's private part and the package body may contain all the above, but one or the other (or both) will contain variable declarations representing the abstraction's state.

Consider the following ADM version of the package Integer_Stacks, now renamed to Integer_Stack for reasons we will discuss shortly. In this version we declare the state in the package body.

```
package Integer_Stack is
  procedure Push (Item : in Integer);
  procedure Pop (Item : out Integer);
  function Empty return Boolean;
  Capacity : constant := 100;
end Integer_Stack;
```

(continues on next page)

(continued from previous page)

```

package body Integer_Stack is
  Values : array (1 .. Capacity) of Integer;
  Top    : Integer range 0 .. Capacity := 0;
  procedure Push (Item : in Integer) is
  begin
    -- ...
    Top := Top + 1;
    Values (Top) := Item;
  end Push;
  procedure Pop (Item : out Integer) is ... end Pop;
  function Empty return Boolean is (Top = 0);
end Integer_Stack;

```

Note how the procedure and function bodies directly access the local variables hidden in the package body.

For those readers familiar with programming languages that can declare entities to be "static," the effect is as if the two variables in the package body are static variables.

When using this idiom, there is only one stack (containing values of some type, in this case type Integer). That's why we changed the name of the package from Integer_Stacks, i.e., from the plural form to the singular. It may help to note that what is now the package name was the name of the client's variable name when there was a Stack type involved.

As with the ADT idiom, clients of an ADM can only manipulate the encapsulated state via the visible operations. The difference is that the state to be manipulated is no longer an object passed as an argument to the operations. For illustration, consider the Push procedure. The ADT version requires the client to pass the Stack object intended to contain the new value (i.e., the actual parameter for the formal named This):

```

procedure Push (This : in out Stack; Item : in Integer);

```

In contrast, the ADM version has one less formal parameter, the value to be pushed:

```

procedure Push (Item : in Integer);

```

Here is a call to the ADM version of Push:

```

Integer_Stack.Push (42);

```

That call places the value 42 in the (hidden) array Integer_Stack.Values located within the package body. Compare that to the ADT approach, in which objects of type Stack are manipulated:

```

Answers : Stack;
-- ...
Push (Answers, 42);

```

That call places the value 42 in the (hidden) array Answers.Values, i.e., within the Answers variable. Clients can declare as many Stack objects as they require, each containing a distinct copy of the state defined by the type. In the ADM version, there is only one stack and therefore only one instance of the state variables. Hence the Stack formal parameter is not required.

Rather than declare the abstraction state in the package body, we could just as easily declare it in the package's private section:

```

package Integer_Stack is
  procedure Push (Item : in Integer);
  procedure Pop (Item : out Integer);
  function Empty return Boolean;

```

(continues on next page)

(continued from previous page)

```

    Capacity : constant := 100;
private
    Values : array (1 .. Capacity) of Integer;
    Top    : Integer range 0 .. Capacity := 0;
end Integer_Stack;

```

Doing so doesn't change anything from the client code point of view; just as clients have no compile-time visibility to declarations in the package body, they have no compile-time visibility to the items in the package private part. This placement also doesn't change the fact that there is only one instance of the data. We've only changed where the data are declared. (We will ignore the effect of child packages here.)

Because the two variables are implementation artifacts we don't declare them in the package's visible part.

Note that the private section wasn't otherwise required when we chose to declare the data in the package body.

The ADM idiom applies information hiding to the internal state, like the ADT idiom, except that the state is not in an object declared by the client. Also, like the *Groups of Related Program Units* (page 1184), the implementations of the visible subprograms are hidden in the package body, along with any non-visible entities required for their implementation.

There are no constructor functions returning a value of the abstraction type because the abstraction is not represented as a type. However, there could be one or more initialization procedures, operating directly on the hidden state in the package private part or package body. In the Stack ADM there is no need for them because of the abstraction-appropriate default initial value, as is true of the ADT version.

The considerations regarding selectors/accessors are the same for the ADM as for the ADT idiom, so they are not provided by default. Also like the ADT, so-called *getters* and *setters* are highly suspect and not provided by the idiom by default.

As mentioned, the ADM idiom can be applied to hardware abstractions. For example, consider a target that has a single on-board rotary switch for arbitrary use by system designers. The switch value is available to the software via an 8-bit integer located at a dedicated memory address, mapped like so:

```

Switch : Unsigned_8 with
    Volatile,
    Address => System.Storage_Elements.To_Address (16#FFC0_0801#);

```

Reading the value of the memory-mapped Switch variable provides the rotary switch's current value.

However, on this target the memory at that address is read-only, and rightly so because the only way to change the value is to physically rotate the switch. Writing to that address has no effect whatsoever. Although doing so is a logical error no indication is provided by the hardware, which is potentially confusing to developers. It certainly looks like a variable, after all. Declaring it as a constant wouldn't suffice because the user could rotate the switch during execution.

Furthermore, although mapped as a byte, the physical switch has only 16 total positions, read as the values zero through fifteen. An unsigned byte has no such constraints.

The compiler will enforce the read-only view and the accessor operation can handle the range constraint. The ADM is a reasonable choice because there is only one such physical switch; a type doesn't bring any advantages in this case. The following illustrates the approach:

```

with Interfaces; use Interfaces;
package Rotary_Switch is

```

(continues on next page)

(continued from previous page)

```

subtype Values is Unsigned_8 range 0 .. 15;
function State return Values;
end Rotary_Switch;

```

Clients can then call the function `Rotary_Switch.State` to get the switch's current value, as a constrained subtype. The body will handle all the details.

```

with System.Storage_Elements; use System.Storage_Elements;
package body Rotary_Switch is
  Switch : Unsigned_8 with Volatile, Address => To_Address (16#FFC0_0801#);
  function State return Values is
  begin
    if Switch in Values then
      return Switch;
    else
      raise Program_Error;
    end if;
  end State;
end Rotary_Switch;

```

The range check in the function body might be considered over-engineering because the switch is a physical device that cannot have more than 16 values, but physical devices have a habit of springing surprises. Note that *attribute Valid* (page 388) would not be useful here because there are no invalid bit patterns for an unsigned integer. If, on the other hand, we were working with an enumeration type, for example, then *'Valid* would be useful.

32.3 Pros

In terms of abstraction and information hiding, the ADM idiom provides the same advantages as the ADT idiom: clients have no visibility to representation details and must use the operations declared in the package to manipulate the state. The compiler enforces this abstract view. The ADM also has the ADT benefit of knowing where any bugs could possibly be located. If there is a bug in the behavior, it must be in the one package defining the abstraction itself. No other code would have the compile-time visibility necessary.

In addition, less source code text is required to express the abstraction.

32.4 Cons

The disadvantage of the ADM is the lack of flexibility.

An ADM defines only one abstraction instance. If more than one becomes necessary, the developer must copy-and-paste the entire package and then change the new package's unit name. This approach doesn't scale well.

Furthermore, the ADM cannot be used to compose other types, e.g., as the component type in an array or record type. The ADM cannot be used to define the formal parameter of a client-defined subprogram, cannot be dynamically allocated, and so on.

But if one can know with certainty that only one thing is ever going to be represented, as in the hardware rotary switch example, the ADM limitations are irrelevant.

PROGRAMMING BY EXTENSION

33.1 Motivation

When declaring entities in a package, developers should ensure that the client view — the package visible part — contains no implementation artifacts. Doing so is important conceptually, but also practically, because any declarations visible to clients inevitably will be used by clients and, as a result, will become permanent fixtures because removal would cause expensive changes in the client code.

The intended client API declarations must be in the package visible part, of course. The question, then, is whether to declare implementation artifacts in the package private part or in the package body. Those are the two parts of a package that do not make declarations compile-time visible to client code.

Some of these entities must be declared in the package private part because they are required in the declaration of some other entity appearing in that part. For example, when using the *ADT idiom* (page 1187), an ancillary type might be required for the completion of the private type. That was the case with the *ADT version* (page 1189) of the `Integer_Stacks` package, repeated here for convenience:

```
package Integer_Stacks is
  type Stack is private;
  -- ...
  Capacity : constant := 100;
private
  type Content is array (1 .. Capacity) of Integer;
  type Stack is record
    Values : Content;
    Top    : Integer range 0 .. Capacity := 0;
  end record;
end Integer_Stacks;
```

The array type `Content` is required for the `Stack` record component because anonymously-typed array components are illegal. Clients have no business using the type `Content` directly so although it would be legal to declare it in the public part, declaration in the private part is more appropriate.

Likewise, a function called to provide the default initial value for a private type's component must be declared prior to the reference. If the function is truly only part of the implementation, we should declare it in the package private part rather than the public part.

In contrast, there may be implementation-oriented entities that are referenced only in the package body. They could be declared in the package body but could alternatively be declared in the package declaration's private part. Those are the entities (declarations) in question for this idiom.

For a concrete example, here is an elided *ADM version of the stack abstraction* (page 1194), with the stack state declared in the package body:

```
package Integer_Stack is
  procedure Push (Item : in Integer);
  procedure Pop  (Item : out Integer);
  function Empty return Boolean;
  Capacity : constant := 100;
end Integer_Stack;

package body Integer_Stack is
  Values : array (1 .. Capacity) of Integer;
  Top    : Integer range 0 .. Capacity := 0;
  procedure Push (Item : in Integer) is
  begin
    -- ...
    Top := Top + 1;
    Values (Top) := Item;
  end Push;
  procedure Pop (Item : out Integer) is ...
  function Empty return Boolean is ...
end Integer_Stack;
```

We could add the private part to the package declaration and move the state of the [ADM](#) (page 1193) — the two variables in this case — up there without any other changes. The subprogram bodies have the same visibility to the two variables either way. (There is no requirement for the Content type because Values is not a record component; anonymously-typed array objects are legal.) From the viewpoint of the language and the abstraction, the location is purely up to the developer.

33.2 Implementation(s)

When you have a choice of placement, putting the state in either the package private part or the package body is reasonable, but only one of the two is amenable to future requirements.

Specifically, placement in the private part of the package allows *programming by extension*³⁴⁶ via hierarchical *child* packages. Child packages can be written immediately after the *parent* package but can also be written years later, thus accommodating changes due to new requirements.

Programming by extension allows us to extend an existing package's facilities without having to change the existing package at all. Avoiding source code changes to the existing package is important because doing so might be very expensive. In certified systems, the changed package would require re-certification, for example. Changes to the parent package are avoidable because child packages have compile-time visibility to the private part of the ancestor package (actually the entire ancestor package hierarchy, any of which could be useful). Thus, the extension in the child package can depend on declarations in the existing parent package's private part.

Therefore, if the developer can know, with certainty, that no visibility beyond the one package will ever be appropriate, the declaration should go in the package body. Otherwise, it should go in the package private part, just in case an extension becomes necessary later.

Using our ADM stack example, we could move the state from the package body to the private part:

```
package Integer_Stack is
  procedure Push (Item : in Integer);
  procedure Pop  (Item : out Integer);
  function Empty return Boolean;
  Capacity : constant := 100;
```

(continues on next page)

³⁴⁶ J. Barnes. *Programming In Ada 95*. Addison-Wesley, 1998.

(continued from previous page)

```

private
  Values : array (1 .. Capacity) of Integer;
  Top    : Integer range 0 .. Capacity := 0;
end Integer_Stack;

```

Note that the private part was not otherwise required by the language in this example.

With that change, a new child package could be created with extended functionality:

```

package Integer_Stack.Utils is
  procedure Reset;
end Integer_Stack.Utils;

package body Integer_Stack.Utils is
  procedure Reset is
  begin
    Top := 0;
  end Reset;
end Integer_Stack.Utils;

```

These child packages are not client code, they contain extensions to the existing abstraction. Hence they are part of what may be considered a subsystem consisting of the original package and the new child package. The child package contains an extension of the abstraction defined by the parent package, so the child is directly related. Given that characterization of child packages we can say that the parent package private part is not visible to client code and, therefore, does not represent a *leak* of implementation details to clients.

33.3 Pros

We can extend an abstraction without changing the source code defining that abstraction, thereby meeting new requirements without incurring potentially expensive redevelopment.

33.4 Cons

Clients could abuse the hierarchical package visibility rules by creating a child package that doesn't really extend the existing package abstraction.

Abuse of the visibility rules allows child packages that can break the abstraction. For example, if we only change the name of procedure Reset in package Integer_Stack.Utils to Lose_All_Contained_Data then the routine has a rather different complexion.

Similarly, abuse of the visibility rules allows child packages that can indirectly leak state from the parent package. For example:

```

package Integer_Stack.Leaker is
  function Current_Top return Integer;
end Integer_Stack.Leaker;

package body Integer_Stack.Leaker is
  function Current_Top return Integer is (Top);
end Integer_Stack.Leaker;

```

We could do that without even requiring a package body, using an expression function for the completion:

```

package Integer_Stack.Leaker is
  function Current_Top return Integer;

```

(continues on next page)

(continued from previous page)

```
private
  function Current_Top return Integer is (Top);
end Integer_Stack.Leaker;
```

The function must be completed in the private part because that is where compile-time visibility to the parent begins within a child package.

Code reviews are the only way to detect these abuses, although detection of potential cases could be automated with an analysis tool such as [Libadalang](https://github.com/AdaCore/libadalang)³⁴⁵.

33.5 Relationship With Other Idioms

We assume the use of the *Abstract Data Type* (page 1187) or *Abstract Data Machine* (page 1193) idioms for the existing package abstraction, as well as for the child package.

33.6 Notes

This guideline will already be used when developing a subsystem (a set of related packages in an overall hierarchy) as a structuring approach during initial development. The idiom discussed here is yet another reason to use the private part, but in this case for the sake of the future, rather than initial, development.

The very first version of Ada (Ada 83) did not have hierarchical library units so, typically, anything not required in the private part was declared in the package body. Declaring them in the private part would only clutter the code that had to be there, without any benefit. The author's personal experience and anecdotal information confirms that after Ada 95 introduced hierarchical library units, some declarations in existing package bodies tended to "percolate up" to the package declarations' private parts.

³⁴⁵ <https://github.com/AdaCore/libadalang>

CONSTRUCTOR FUNCTIONS FOR ABSTRACT DATA TYPES

34.1 Motivation

In languages supporting object-oriented programming (OOP), including Ada, *constructors* are not inherited when one type is derived from another. That's appropriate because, in general, they would be unable to fully construct values for the new type. The purpose of this idiom is to explain how Ada defines constructors, how the language rules prevent constructor inheritance, and how to design the constructor code in light of those rules.

Ada uses tagged types to fully support dynamic OOP. Therefore, in the following, a *derived type* refers to a tagged type that is declared as a so-called *type extension* — a form of inheritance — based on some existing *parent* tagged type. The extension consists of additional components and/or additional or changed operations beyond those inherited from the existing parent type.

This discussion assumes these tagged types are declared in packages designed using the [Abstract Data Type](#) (page 1187) (ADT) idiom. We strongly recommend the reader be comfortable with that idiom before proceeding.

As abstract data types, the parent type is a private type, and the derived type is a *private extension*. A private extension is a type extension declaration that does not reveal the components added, if any. The parent type could itself be an extended type, but the point is that these types will all be private types one way or another. Declarations as private types and private extensions are not required by the language for inheritance, but as argued in the ADT idiom discussion, doing so is recommended in the strongest terms. OOP doesn't change that, and in fact the encapsulation and information hiding that are characteristic of the ADT idiom are foundational principles for OOP types.

For an example of a private extension, given a tagged type named `Graphics.Shape` one can declare a new type named `Circle` via type extension:

```
type Circle is new Graphics.Shape with private;
```

This declaration will be in the public part of a package, but, as a private type extension, the additional components are not compile-time visible to client code, conforming to ADT requirements. That's what the reserved word `private` indicates in the type declaration.

Instead of a distinct constructor syntax, Ada uses regular functions to construct objects. Specifically, so-called *constructor functions* are functions that return an object of the type.

```
type Circle is new Graphics.Shape with private;  
function New_Circle (Radius : Float) return Circle;
```

Like any function there may be formal parameters specified, but not necessarily.

Functions and procedures that manipulate objects of the private type are *primitive operations* for the type if they are declared in the same package as the type declaration itself. For procedures, that means they have formal parameters of the type. For functions, that

means they either have formal parameters of the type, or return a value of the type, or both.

Declaration within the same package as the type itself provides the compile-time visibility to the type's representation required to implement the subprograms.

Other operations might be declared in the same package too, but if they do not manipulate or return values of the type they are not primitive operations for the type. (Their location in that package is somewhat suspect and should be reviewed explicitly.)

Primitive operations, and only primitive operations, are inherited during type derivation.

If you think in terms of Abstract Data Types all these rules make sense.

Now, here's the rub.

Constructor functions require that same compile-time visibility so the intuitive approach will be to declare them in the same package declaration as the type. As a result, they will be primitive operations for that type.

However, that means that the constructor functions will be inherited, contrary to the expectation for constructors. Therefore, Ada has rules specific to primitive constructor functions that have the effect of preventing their inheritance.

The explanation and illustration for these rules first requires explanation of the word *abstract*. We mentioned above that the package enclosing the type will be designed with the *Abstract Data Type* (page 1187) idiom. In that idiom *abstract* means that the type represents an abstraction. (See that section for the details.)

The term *abstract* also has a meaning in OOP, one that is unrelated to an ADT. In OOP, an *abstract* type is one that defines an interface but at most a partial implementation. As such, the type can serve as the ancestor type for derived types but cannot be used to declare objects. An abstract type in Ada includes the reserved word **abstract** in the declaration. For example:

```
type Foo is abstract tagged private;
```

Similarly, subprograms can be abstract. These again define an interface, via the subprogram formal parameters and result type, but are not callable units. In Ada these too include the word **abstract** in their declarations, for example:

```
procedure Do_Something (This : in out Foo) is abstract;
```

Now we can explain how Ada prevents constructor inheritance.

Whenever a tagged type is extended, all inherited constructor functions automatically become abstract functions for the extended type, just as if they were explicitly declared *abstract*.

However, only abstract types can legally have abstract primitive operations. Concrete types may not, so that we can never dynamically dispatch to a subprogram without an actual implementation.

Therefore, unless the extended child type is itself abstract, the type extension will be illegal. The compiler will reject the declaration of the child type, thus preventing this inappropriate constructor inheritance.

For an example, both to illustrate the code and the Ada rules, consider this simple package declaration that presents the tagged private type `Graphics.Shape`:

```
package Graphics is
  type Shape is tagged private;
  function Make (X, Y : Float) return Shape;
  ...
private
```

(continues on next page)

(continued from previous page)

```

type Shape is tagged record
  X : Float := 0.0;
  Y : Float := 0.0;
end record;
end Graphics;

```

Note in particular the primitive constructor function named `Make` that constructs a value of type `Shape`.

Because type `Shape` is tagged, other types can extend it:

```

with Graphics;
package Geometry is
  type Circle is new Graphics.Shape with private; -- a private extension
  -- ...
private
  type Circle is new Graphics.Shape with record
    Radius : Float;
  end record;
end Geometry;

```

Type `Circle` automatically inherits the components and primitive operations defined by type `Shape`, including the constructor function `Make`. No additional declarations are required in order to inherit these operations or components. The inherited operations are now primitive operations for the new type.

Inherited primitive operations have an unchanged formal parameter and result-type profile, except for the controlling parameter type name, so although `Make` now returns a `Circle` object, the function still only has parameters for the `X` and `Y` components. Hence this version of `Make` could not set the `Radius` component in the returned `Circle` value to anything other than some default.

Therefore, to prevent this inherited function from being available, two Ada rules come into play. The first rule specifies that the implicit function is inherited as if declared explicitly abstract:

```

function Make (X, Y : Float) return Circle is abstract;
-- as actually inherited, implicitly

```

Note the reserved word **abstract** in the implicit function declaration. This declaration doesn't actually appear in the source code because all the inherited primitive operations are implicitly declared.

Another rule specifies that only abstract types can have abstract primitive subprograms. Type `Circle` is not abstract in this sense, therefore the combination of those two rules makes the `Circle` type extension illegal. Package `Geometry` will not compile successfully.

Failing to compile is safe — it prevents clients from having a callable function that in general cannot suffice — but requires an alternative so that sufficient constructor functions are possible.

Therefore, a general design idiom is required for defining constructor functions for concrete tagged Abstract Data Types.

34.2 Implementation(s)

The general approach uses functions for constructing objects but prevents these functions from being inherited. The problem is thus circumvented entirely.

To prevent their being inherited, the implementation prevents the constructor functions from being primitive operations. However, these functions require compile-time visibility

to the parent type's representation in order to construct values of the type, as this typically involves assigning values to components in the return object. The alternative approach must supply the compile-time visibility that primitive operations have.

Therefore, the specific implementation is to declare constructor functions in a separate package that is a *child* of the package declaring the tagged type. This takes advantage of the *hierarchical library units* capability introduced in Ada 95.

Operations declared in a child package are not primitive operations for the type in the parent package, so they are not inherited when that type is extended. Consequently they do not become abstract.

In addition, the required visibility to the parent type's representation in the private part will be available to the functions' implementations because the private part and body of a child package have compile-time visibility to the parent package's private part.

In this idiom, any package declaring a tagged type, either directly or by type extension, will have a *constructors* child package if constructors are required. For example:

```
package Graphics.Constructors is
  function Make (X, Y : Float) return Shape;
end Graphics.Constructors;
```

and similarly, for Circle:

```
package Geometry.Constructors is
  function Make (X, Y, R : Float) return Circle;
end Geometry.Constructors;
```

Each of these two package declarations will have a package body containing the body of the corresponding function. In fact such packages can declare as many constructor functions as required, overloaded or not.

Clients that want to use a constructor function will specify the constructor package in the context clauses for their units, as usual. The constructor package body for an extended type might very well do so itself, as shown below:

```
with Graphics.Constructors; use Graphics.Constructors;
package body Geometry.Constructors is
  function Make (X, Y, R : Float) return Circle is
    (Circle'(Make (X, Y) with Radius => R));
end Geometry.Constructors;
```

Of course, the name "Constructors" is not required for the child packages. It could be "Ctors", for example (a name common in C++), or something else. But whatever the choice, regularity enhances comprehension so the same child package name should be used throughout.

34.3 Pros

The issue is sidestepped entirely, and as an additional benefit, the parent packages are that much simpler because the constructor function declarations and bodies are no longer present there. The *constructors* child packages themselves will be relatively simple since they contain only the constructor functions and any ancillary code required to implement them. Simpler code enhances comprehension and correctness.

Having the constructors declared in separate packages applies the principle of Separation of Concerns, between the code defining the type's semantics and the code for constructing objects of the type. This principle also enhances comprehension.

34.4 Cons

There will be a child package for each tagged type that requires constructors, hence more packages and files (assuming one unit per file, which is desirable in itself, even if not required by the language).

Some developers might argue for having fewer files, presumably containing larger units. In the author's experience larger units make comprehension, and therefore correctness, unjustifiably difficult if smaller units are possible. Some units are unavoidably large and complicated but often we can achieve relative simplicity.

For those developers, however, the constructor package could be declared instead as a nested package located within the package defining the tagged type. Doing so would achieve the same effect as using a child package because the contained functions would not be primitive. Therefore, they would not be inherited.

This alternative would reduce the number of files back to the minimum. However, the defining package would be relatively more complicated because of this nested package. Note that the nested package declaration would require a nested package body too.

In short, the alternative reduces the number of files at the cost of additional unit complexity. (If the issue with the larger number of files is difficulty in locating individual entities of interest, any decent IDE will make doing so trivial.)

The alternative also loses the distinction between clients that use objects of the type and clients that create those objects, because, with the child package approach, the latter will be the only clients that have context clauses for the constructor packages.

34.5 Relationship With Other Idioms

N/A

34.6 Notes

For those interested, in this section we provide a discussion of alternatives to the implementation presented, and why they are inadequate.

Changing the behavior of an inherited operation requires an explicit conforming subprogram declaration and therefore a new subprogram body for that operation. This change is known as *overriding* the inherited operation.

Package Geometry could declare a function with the additional parameters required to fully construct a value of the new type. In this case the new constructor would include the Radius parameter:

```
function Make (X, Y, Radius : Float) return Circle;
```

But such a function would not be overriding for the inherited version because the parameter and result type profile would be different. This function Make would overload the inherited function, not override it. The inherited function remains visible, as-is.

In fact, we could even have the compiler confirm that this is not an overriding function by declaring it so:

```
not overriding function Make (X, Y, Radius : Float) return Circle;
```

In general, specifying that a subprogram is not overriding is less convenient than specifying that it is overriding. We only do so in these examples to make everything explicit.

Because that new function is not overriding, the inherited version remains implicitly abstract and the type extension remains illegal. Developers could also override the inherited function, which would make the code legal, but as we have said such a function cannot properly construct values in general, and might be called accidentally. For example:

```
with Graphics;
package Geometry is
  type Circle is new Graphics.Shape with private;

  overriding function Make (X, Y : Float) return Circle;

  not overriding function Make (X, Y, Radius : Float) return Circle;
  -- overloading

  ...
private
  -- ...
end Geometry;
```

Although the overridden Make does not have a Radius parameter and could only assign some default to that component, if that default is reasonable then the overridden function could be called on purpose, i.e., not accidentally. That's not a general approach, however.

Alternatively, developers could use procedures as their constructors, with a mode-out parameter for the result. The procedure would not become implicitly abstract in type extensions, unlike a function.

```
package Graphics is
  type Shape is tagged private;
  procedure Make (Value : out Shape; X, Y : in Float);
private
  -- ...
end Graphics;
```

And then the client extension would inherit the procedure:

```
with Graphics;
package Geometry is
  type Circle is new Graphics.Shape with private;
  -- procedure Make (Value : out Circle; X, Y : in Float); -- inherited
private
  -- ...
end Geometry;
```

However, although now legal, the inherited procedure would not suffice, lacking the required parameter for the Radius component.

Developers might then add an overloaded version with the additional parameter:

```
with Graphics;
package Geometry is
  type Circle is new Graphics.Shape with private;

  -- procedure Make (Value : out Circle; X, Y : in Float);
  -- inherited

  not overriding procedure Make (Value : out Circle; X, Y, R : in Float);
  -- not inherited
private
  -- ...
end Geometry;
```

But the same issues arise as with functions. Clients might accidentally call the wrong proce-

dure, i.e., the inherited routine that doesn't have a parameter for the Radius. That routine would not even mention the Radius component, much less assign a default value, so it would have to be overridden in order to do so. This too is not a general approach.

CONTROLLING OBJECT INITIALIZATION AND CREATION

35.1 Motivation

Developers are responsible for ensuring that no uninitialized objects are read in Ada programs. Default initialization is a good way to meet this requirement because it is guaranteed to happen and requires no actions on the part of the client code. But of the many kinds of types provided by Ada, only access types have a language-defined default initial value. Fortunately, Ada supports user-defined default initialization for user-defined types.

Default initialization is conveniently expressed, especially because components of record types can have default initial values. Record types are perhaps the most commonly used non-numeric type in the language. Sometimes a given type was *wrapped* inside a record type purely for the sake of default component initialization, e.g., numeric types. That wrapping approach is less common than in earlier versions of the language, given the comparatively more recent aspect `Default_Value` for scalar types, and `Default_Component_Value` for scalar array components.

These facilities are often sufficient to express an abstraction's initial state. For example, we can expect that container objects will be initially empty. Consider a bounded stack ADT. The representation is likely a record type containing an array component and a `Top` component indicating the index of the last array component used. We can default initialize objects to the empty state simply by setting `Top` to zero in the record component's declaration:

```
type Content is array (Positive range <>) of Element;
type Stack (Capacity : Positive) is record
  Values : Content (1 .. Capacity);
  Top    : Natural := 0;
end record;
```

For an unbounded container such as a simple binary tree, if the representation is an access type, the automatic default value `null` initializes `Tree` objects to the empty state.

```
package Binary_Trees is
  type Tree is limited private;
  ...
private
  type Leaf_and_Branch is record ...
  type Tree is access Leaf_and_Branch;
  ...
end Binary_Trees;
```

In both cases, simply declaring an object in the client code is sufficient to ensure it is initially empty.

However, not all abstractions have a meaningful default initial state. Default initialization will not suffice to fully initialize objects in these cases, so explicit initialization is required.

An explicit procedure call could be used to set the initial state of an object (passed to a mode-out parameter), but there is no guarantee that the call will occur and no way to force

a client to make it.

In contrast, the declaration of the object is guaranteed to occur, and as part of the declaration the object can be given an explicit initial value. The initial value can be specified by a literal for the type, by the value of another object of that type, or by the value of that type returned from a function call.

```
declare
  X : Integer := Some_Existing_Integer_Object;
  Prompt : constant String := "Name? ";
  Reply : constant String := Response (Prompt);
begin
  ...
end;
```

The initial value can also specify constraints, if required. In the code above, the object Prompt has a lower bound of **Positive**'First and an upper bound set to the length of the literal. The specific bounds of Reply are determined by the function, and need not start at **Positive**'First.

An object cannot be used before it is declared. Since this explicit initial value is part of the declaration, the object cannot be read before it is initialized. That fact is the key to the implementation approaches.

However, although the object declaration is guaranteed to occur, explicit initialization is optional. But unlike a procedure call, we can force the initial value to be given. There are two ways to force it, so there are two implementations presented.

In addition, a specific form of explicit initialization may be required because not all forms of initialization are necessarily appropriate for a given abstraction. Imagine a type representing a thread lock, implemented in such a way that default initialization isn't an option. Unless we prevent it, initialization by some other existing object will be possible:

```
declare
  X : Thread_Lock := Y;    -- Y is some other Thread_Lock object
begin
  -- ...
end;
```

This would amount to a copy, which might not make sense. Imagine the lock type contains a queue of pending callers...

More generally, if a type's representation includes access type components, initialization by another object will create a shallow copy of the designated objects. That is typically inappropriate.

Using an existing object for the initial value amounts to a complete copy of that other object, perhaps more of a copy than required. For example, consider a bounded container type, e.g., another stack, backed by an array and an index component named Top. At any time, for any stack, the contained content is in the slice of the array from 1 up to Top. Any array component at an index greater than Top has a junk value. Those components may never even have been assigned during use. Now consider the declaration of a Stack object, A, whose initial value is that of another existing Stack named B.

```
A : Stack := B;
```

The entire value of B is copied into A, so B.Top is copied to A.Top, which makes sense. But likewise, the entire array in B will be copied to the array in A. For a stack with a large backing array that might take a significant amount of time. If B is logically full then the time required for the full array copy is unavoidable. But if only a few values are contained by B, the hit could be avoided by only copying up to Top.

And of course, the initial value might require client-specific information.

Calling a *constructor function* (page 1203) for the initial value would be the right approach in these cases, returning an object of the type. The function might even take an existing object as a parameter, creating a new object with only the necessary parts copied.

Therefore, for some abstractions, not only do we need to guarantee explicit object initialization, we may also need to restrict the form of initial value to a function call.

The other purpose of the idiom is controlling, for some type, whether object creation itself is to be allowed by clients. As you will see, controlling object initialization can be used to control object creation.

Preventing object creation is not typical but is not unknown. The *singleton design pattern*³⁴⁷ is an example, in which a type is defined but corresponding object creation by clients is not intended. Instead, the abstraction implementation creates a single object of the type. The abstraction is a type, rather than an *ADM* (page 1193), for the sake of potential extension via inheritance. We will illustrate this design pattern and implementation using a real-world hardware device.

35.2 Implementation(s)

There are two ways to force an explicit initial value as part of an object declaration. One is a matter of legality at compile-time so it is enforced by the compiler. The other is enforced by a run-time check.

Note that both approaches are type-specific, so when we say *objects* we mean objects of a type that has been designed with one of these two idiom implementations. Neither implementation applies to every object of every type used in the client code. (SPARK, a formal language based closely on Ada, statically ensures all objects are initialized before read.)

The *ADT idiom* (page 1187) describes Ada *building blocks* that developers can use to compose types with semantics that we require. We can declare a type to be private, for example, so that the implementation is not compile-time visible to clients.

In addition to private types, we can decorate a type declaration with the reserved word **limited** so that assignment is not allowed (among other things) for client objects of the type. We can combine the two building blocks, creating a type that is both private and limited.

Throughout this discussion we will assume that these designs are based on *Abstract Data Types* (page 1187), hence we assume the use of private types. That's a general, initial design assumption but in this case private types are required by the two idiom implementations. The types are not necessarily limited as well, but in one situation they will be limited too. But in both implementations the primary types will be private types.

35.2.1 Compile-Time Legality

We can combine the private type and limited type building blocks with another, known as *unknown discriminants*, to force explicit object initialization by clients, to control the form of explicit initialization, and, when required, to control client object creation itself. Limited and private types are fairly common building blocks, but *unknown discriminants* are less common so we will first explain them, and then show how to utilize the combinations for this idiom.

Discriminants are useful for our purpose because types with discriminants are *indefinite types* (under certain circumstances). Indefinite types do not allow object declarations without also specifying some sort of constraints for those objects. Unconstrained array types, such as **String**, are good examples. We cannot simply declare an object of type **String** without also specifying the array bounds, one way or another:

³⁴⁷ https://en.wikipedia.org/wiki/Singleton_pattern


```
with Ada.Text_IO; use Ada.Text_IO;
procedure Initialization_Demo is
  S1 : String (1 .. 11) := (others => ' ');
  S2 : String := "Hello World";
  S3 : String := S1;
begin
  Put_Line (''' & S1 & '');
  Put_Line (''' & S2 & '');
  Put_Line (''' & S3 & '');
end Initialization_Demo;
```

In the code above, **String** objects S1, S2, and S3 all have the same constraints: a lower bound of **Positive**'*First* and an upper bound of 11. S1 gives the bounds directly, whereas S2 and S3 take their constraints from their initial values. A function that returned a **String** value would suffice for the initial value too and would thus serve to specify the array bounds. There are other ways to specify a constraint as well, but we can ignore them in this idiom because the building blocks we'll use preclude them.

Types with discriminants are indefinite types unless the discriminants have default values. That fact will not apply in this idiom because of the characteristics of the building blocks. You will see why in a moment. The important idea is that we can leverage the object constraint requirements of indefinite types to force explicit initialization on declarations.

Discriminants come in two flavors. So-called *known* discriminants are the most common. These discriminants are known in the sense that they are compile-time visible to client code. Clients then have everything needed for declaring objects of the corresponding type. For example, here is the type declaration for a bounded stack ADT:

```
type Stack (Capacity : Positive) is private;
```

In the above, Capacity is the physical number of components in the array backing the bounded implementation. Clients can, therefore, have different objects of the type with different capacities:

```
Trays : Stack (Capacity => 10);
Operands : Stack (100);
```

The existence of Capacity is known to clients via the partial view, so the requirement for the constraint is visible and can be expressed.

In contrast, types may have *unknown discriminants* in the client's view. The syntax reflects their confidential nature:

```
type Foo (<>) is private;
```

The parentheses are required as usual, but the *box* symbol appears inside, instead of one or more discriminant declarations. The box symbol always indicates *not specified here* so in this case no discriminants are included in the view. There may or may not be discriminants in the full view, but client's don't have compile-time visibility to that information because the type is private.

Unknown discriminants can be specified for various kinds of types, not only private types. See the [Notes section](#) (page 1228) for the full list. That said, combining them with private type declarations, or private type extension declarations, is the most common usage when composing abstraction definitions. For example:

```
package P is
  type Q (<>) is private;
private
  type Q is range 0 .. 100;
end P;
```

Clients of package P must use type Q as if Q requires discriminant constraints, even though clients don't have compile-time visibility to whatever constraints are actually required, if any. In the above, Q is just an integer type in the full view. No constraint is required to create objects of type Q, but clients cannot take advantage of that fact because they only have the partial view. Only the package private part, the package body, and child units have the visibility required to treat Q as an integer type.

Q might actually be completed as an indefinite type, but the constraint required need not be a discriminant constraint. In the following, objects of type Q require an array bounds constraint:

```
package P is
  type Q (<>) is private;
private
  type Q is array (Positive range <>) of Integer;
end P;
```

Code with the full view must respect the index bounds requirement, but the semantics of the partial view remain the same.

As illustrated, the consequence of combining indefinite types with private types is that, when declaring objects, clients must express a constraint but cannot do so directly. The constraints must instead be provided by the initial value. Hence, for these types, the initial value is now a requirement that the compiler enforces on client object declarations.

But because the type is private, the initial value cannot be specified by a literal. Instead, the initial value must be either an existing object of the type, or the result of a call to a function that returns an object of the type.

Consider the following:

```
package P is
  type Q (<>) is private;
  function F return Q;
private
  type Q is range 0 .. 100;
end P;

package body P is
  function F return Q is (42);
  -- since that is the answer to everything...
end P;

with P;
procedure Demo is
  Obj1 : P.Q; -- not legal, requires initial value for constraint
  Obj2 : P.Q := 42; -- not legal, per client's partial view
  Obj3 : P.Q := P.F;
  Obj4 : P.Q := Obj3;
begin
  null;
end Demo;
```

The declaration for Obj1 is illegal because no constraint is provided. Because P.Q is also private, the declaration of Obj2 is illegal because clients don't have the full view supporting integer usage. But the initial value can be provided by a function result (Obj3), thereby also specifying the required constraint. And an existing object can be used to give the constraints to other objects during their declarations (Obj4). Explicit client initialization in these two ways is required by the compiler for indefinite private types.

But as illustrated by the spin-lock example, initialization by an existing object is not always appropriate. We can restrict the initial value to a function call result by making the type limited as well as private and indefinite. Then only constructor functions can be used legally

for the initial values, and the compiler will require them to be called during object declarations (e.g., `Obj3` above). That's what we'd do for the spin-lock type. We'd make the type limited in the completion too, to prevent copying in any form, including the function result. (The function result would then be built in place instead of copied.)

To recap, the primary purpose of the idiom, for a given type, is to ensure that clients initialize objects of that type as part of the object declarations. In this first implementation we meet the requirement by composing the type via building blocks that:

1. require a constraint to be given when declaring any object of the type, and
2. require an initial value to give that constraint, and
3. allow only objects and function call results as the initial values, and
4. when necessary, allow only function call results to be used for the initial values.

The compiler will reject declarations that do not adhere to these rules. Explicit initialization in the client code is thus guaranteed.

For a concrete example, consider a closed loop process controller, specifically a [proportional-integral-derivative \(PID\) controller](#)³⁴⁸. A PID controller examines the difference between an intended value, such as the desired speed of your automobile, and the current value (the actual speed). In response to that difference the controller increases or decreases the throttle setting. This measurement and resulting control output response happens iteratively at some rate. This is a sophisticated ADT, and explaining how a PID controller actually works is beyond the scope of this document. There are numerous web sites available that describe them in detail. What you should know for our purpose is that they are used to control physical processes, such as your car's cruise control system, that affect our lives directly. Ensuring proper initialization is part of ensuring correct use.

The PID controller must be explicitly initialized because there is no default initial state that would allow subsequent safe use. Only a partial meaningful state can be defined by default. Specifically, a PID controller can be enabled and disabled by the user (the external process control engineer) at arbitrary times. We can define default initialization such that the objects are initially in the disabled state. When disabled, the output computation actually affects nothing, so starting from that state would be safe. However, there is nothing to prevent the user from enabling the controller object without first configuring it. Configuring the various parameters is essential for safe and predictable behavior.

To address that problem, we could add operation preconditions requiring the object to be in some *configured* state, but that isn't always appropriate. Such a precondition would just raise an exception, which isn't in the use-cases. (Statically proving prior configuration in the client code would be a viable alternative, but that's also beyond the scope of this document.)

Therefore, default initialization doesn't really suffice for this ADT. We need to force initialization (configuration) during object creation so that enabling the ADT output will always be safe. This idiom implementation does exactly that.

The following is a cut-down version of the package declaration using this idiom implementation, with some operations and record components elided for the sake of simplicity. In the full version the unit is a generic package for the sake of not hard-coding the floating point types. We use a regular package and type **Float** here for convenience. The full version is here:

- [AdaCore/Robotics_with_Ada/src/control_systems \(GitHub\)](#)³⁴⁹

```
package Process_Control is
```

```
    type PID_Controller (<>) is tagged limited private;
```

(continues on next page)

³⁴⁸ https://en.wikipedia.org/wiki/Proportional-integral-derivative_controller

³⁴⁹ https://github.com/AdaCore/Robotics_with_Ada/blob/master/src/control_systems/

(continued from previous page)

```

type Bounds is record
    Min, Max : Float;
end record with
    Predicate => Min < Max;

type Controller_Directions is (Direct, Reversed);

type Millisecond_Units is mod 2**32;

subtype Positive_Milliseconds is
    Millisecond_Units range 1 .. Millisecond_Units'Last;

function Configured_Controller
    (Proportional_Gain : Float;
     Integral_Gain     : Float;
     Derivative_Gain   : Float;
     Invocation_Period : Positive_Milliseconds;
     Output_Limits     : Bounds;
     Direction         : Controller_Directions := Direct)
return PID_Controller;

procedure Enable
    (This      : in out PID_Controller;
     Process_Variable : Float;  -- current input value from the process
     Control_Variable : Float); -- current output value

procedure Disable (This : in out PID_Controller);

procedure Compute_Output
    (This      : in out PID_Controller;
     Process_Variable : Float;          -- the input, Measured Value/Variable
     Setpoint      : Float;
     Control_Variable : in out Float); -- the output, Manipulated Variable

    -- ...

function Enabled (This : PID_Controller) return Boolean;

private

    type PID_Controller is tagged limited record
        -- ...
        Enabled : Boolean;
    end record;

end Process_Control;

```

As you can see, the PID controller type is indefinite limited private:

```
type PID_Controller (<>) is tagged limited private;
```

It is also tagged, primarily for the sake of the distinguished receiver call syntax. We don't really expect type extensions in this specific ADT, although nothing prevents them.

Therefore, the language requires an initial value when creating objects of the type, and because the type is limited, a function must be used for that initial value. The compiler will not compile the code containing the declaration otherwise. The only constructor function provided is `Configured_Controller` so it is guaranteed to be called. (A later child package could add another *constructor function* (page 1203). For that matter, we probably should have declared this one in a child package. In any case one of them is guaranteed to be called.)

Here is an example declaration taken from the steering control module for an RC car written in Ada³⁵⁰.

The PID controller, named `Steering_Computer`, is declared within the body of a task `Servo` that controls a motor, `Steering_Motor`, in response to requested directions from the remote control. `Steering_Motor` is an instance of an ADT named `Basic_Motors`, and is declared elsewhere. The `Servo` task is declared within the body of a package that contains various values referenced within the task, such as the various PID gain parameters, that are not shown.

```
task body Servo is
  Next_Release      : Time;
  Target_Angle      : Float;
  Current_Angle     : Float := 0.0;
  -- zero for call to Steering_Computer.Enable
  Steering_Power    : Float := 0.0;
  -- zero for call to Steering_Computer.Enable
  Motor_Power       : NXT.Motors.Power_Level;
  Rotation_Direction : NXT.Motors.Directions;
  Steering_Offset    : Float;
  Steering_Computer : PID_Controller :=
    Configured_Controller
      (Proportional_Gain => Kp,
       Integral_Gain    => Ki,
       Derivative_Gain   => Kd,
       Invocation_Period => System_Configuration.Steering_Control_Period,
       Output_Limits     => Power_Level_Limits,
       Direction         => Closed_Loop.Direct);
begin
  Global_Initialization.Critical_Instant.Wait (Epoch => Next_Release);
  Initialize_Steering_Mechanism (Steering_Offset);
  Steering_Computer.Enable (Process_Variable => Current_Angle,
                           Control_Variable => Steering_Power);
  loop
    Current_Angle := Current_Motor_Angle (Steering_Motor) -
      Steering_Offset;
    Target_Angle := Float (Remote_Control.Requested_Steering_Angle);
    Limit (Target_Angle, -Steering_Offset, +Steering_Offset);
    Steering_Computer.Compute_Output
      (Process_Variable => Current_Angle,
       Setpoint         => Target_Angle,
       Control_Variable => Steering_Power);
    Convert_To_Motor_Values (Steering_Power,
                             Motor_Power,
                             Rotation_Direction);
    Steering_Motor.Engage (Rotation_Direction, Motor_Power);

    Next_Release := Next_Release + Period;
    delay until Next_Release;
  end loop;
end Servo;
```

Because `Steering_Computer` must be declared before it can be passed as a parameter, the call to configure the object's state necessarily precedes any other operation (e.g., `Enable`).

35.2.2 Run-Time Checks

Ada 2022 adds another building block, `Default_Initial_Condition` (DIC), that can be used as an alternative to the unknown discriminants used above. We must still have a private type or private type extension, and the type may or may not be limited, but unknown discriminants will not be involved. The compiler would not allow the combination, in fact.

³⁵⁰ <https://blog.adacore.com/making-an-rc-car-with-ada-and-spark>

DIC is an aspect applied to a private type or private extension declaration. Developers use it to specify a developer-defined Boolean condition that will be true at run-time after the default initialization of an object of the type. Specifically, if `Default_Initial_Condition` is specified for a type, a run-time check is emitted for each object declaration of that type that uses default initialization. The check consists of the evaluation of the DIC expression. The exception `Assertion_Error` is raised if the check fails. You can think of this aspect as specifying the effects of default initialization for the type, with a verification at run-time when needed. No check is emitted for those declarations that use explicit initialization.

For example, the following is a partial definition of a Stack ADT. It is only a partial definition primarily because `Pop` is not provided, but other operations would be included as well. Moreover, a fully realistic version would be a generic package. We have used a subtype named `Element` as a substitute for the generic formal type what would have had that name. Note that there is a `Default_Initial_Condition` aspect specifying that any object of type `Stack` is initially empty as a result of default initialization. The *argument* to the function call is the corresponding type name, representing the current instance object, thus any object of the type.

```
package Bounded_Stacks is

  subtype Element is Integer;
  -- arbitrary substitute for generic formal type

  type Stack (Capacity : Positive) is limited private with
    Default_Initial_Condition => Empty (Stack);

  procedure Push (This : in out Stack; Value : Element) with
    Pre => not Full (This),
    Post => not Empty (This);

  function Full (This : Stack) return Boolean;

  function Empty (This : Stack) return Boolean;

private

  type Contents is array (Positive range <>) of Element;

  type Stack (Capacity : Positive) is limited record
    Content : Contents (1 .. Capacity);
    Top      : Natural :=0;
  end record;

  function Full (This : Stack) return Boolean is
    (This.Top = This.Capacity);

  function Empty (This : Stack) return Boolean is
    (This.Top = 0);

end Bounded_Stacks;

package body Bounded_Stacks is

  procedure Push (This : in out Stack; Value : Element) is
  begin
    This.Top := This.Top + 1;
    This.Content (This.Top) := Value;
  end Push;

end Bounded_Stacks;

with Ada.Text_IO; use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```
with Bounded_Stacks; use Bounded_Stacks;

procedure Demo is
  S : Stack (Capacity => 10);
begin
  Push (S, 42);
  Put_Line ("Done");
end Demo;
```

The function `Empty` returns **True** when `Top` is zero, and zero is assigned to `Top` during default initialization. Consequently, `Assertion_Error` is not raised when `Demo` executes because the object `S` was indeed default initialized to the empty state.

We said that when DIC is applied to a type, the run-time check is emitted for all object declarations of that type that rely on default initialization. But suppose the type does not define any default initialization. We can detect these uninitialized objects at run-time if we set the DIC Boolean expression to indicate that there is no default initialization defined for this type. The checks will then fail for those objects. That's the second implementation approach to the initialization requirement.

Specifically, we can express the lack of default initialization by a DIC condition that is hard-coded to the literal **False**. The evaluation during the check will then necessarily fail, raising `Assertion_Error`. Hence, for this type, explicit initialization is guaranteed in a program that does not raise `Assertion_Error` for this cause.

The following is an example of the DIC set to **False**:

```
package P is
  type Q is limited private with
    Default_Initial_Condition => False;

  function F return Q;
private
  type Q is range -1 .. 100;
end P;

package body P is
  function F return Q is (42);
end P;

with Ada.Text_IO; use Ada.Text_IO;

with P; use P;

procedure Main is
  Obj1 : constant Q := F;
  Obj2 : Q; -- triggers Assertion_Error
begin
  Put_Line (Obj1'Image);
  Put_Line (Obj2'Image);
  Put_Line ("Done");
end Main;
```

In the above, `Assertion_Error` is raised by the elaboration of `Obj2` because the DIC check necessarily fails. There is no check on the declaration of `Obj1` because it is initialized, explicitly.

To recap, we can ensure initialization for objects of the type by detecting, during elaboration at run-time, any objects not explicitly initialized.

This approach is sufficient because when elaboration of an object declaration raises an exception, no use of that object is possible. That's guaranteed because the frame containing that declarative part is immediately abandoned and the exception is propagated up to the previous level. A local handler never can apply. But even if there is a matching handler in the previous level, there's really nothing much to be done. Re-entering the frame containing the declaration will raise the exception all over again, necessarily. Thus the code will have to be changed and recompiled, meeting the goal of the idiom.

We can illustrate this assurance using `Storage_Error`. Consider the following program, in which the main procedure calls an inner procedure `P`:

```
with Text_IO; use Text_IO;

procedure Main is

  procedure P (Output : out Float) is
    N : array (Positive) of Float; -- Storage_Error is likely
  begin
    Put_Line ("P's body assigns N's components and uses them");
    -- The following indexes and component values are arbitrary
    -- and used purely for illustration...
    N := (others => 0.0);
    -- other computations and assignments to N ...
    Output := N (5);
  exception
    when Storage_Error =>
      Output := N (1);
  end P;

  X : Float;

begin
  P (X);
  Put_Line (X'Image);
  Put_Line ("Done");
exception
  when Storage_Error =>
    Put_Line ("Main completes abnormally");
end Main;
```

When `Main` calls `P`, the elaboration of the declarative part of `P` almost certainly fails because there is insufficient storage to allocate to the object `P.N`, hence `Storage_Error` is raised. (If your machine can handle the above, congratulations.) Even though procedure `P` has a handler specifically for `Storage_Error`, that handler never applies because the declarative part is immediately abandoned. Instead, the exception is raised in the caller, where it can be caught. This behavior is essential to ensure that problematic objects are not referenced in the local handlers. In the above, the handler in `P` for `Storage_Error` references the object `P.N` to assign the `P.Output` parameter. If that assignment could happen — again, it cannot — what would it mean, functionally? No one knows.

Handling `Storage_Error` is a little tricky anyway. Does the OS give the program a chance to execute a handler? If so, is there sufficient storage remaining to execute the exception handler's statements? In any case you can see the problem that the declaration failure semantics preclude.

Therefore, although the DIC approach is not enforced at compile-time, it is nevertheless sufficient to ensure no uninitialized object of the type can be used.

35.2.3 Preventing Object Creation by Clients

The other idiom requirement is the ability to control object creation itself. The implementation is trivially achieved using an indefinite limited private type: we can prevent client object creation simply by not providing any constructor functions. Doing so removes any means for initializing objects of the type, and since the type is indefinite there is then no way for clients to declare objects at all. The compiler again enforces this implementation.

For a concrete example, we can apply the Singleton design pattern to represent the *time stamp counter* (TSC³⁵¹) provided by x86 architectures. The TSC is a 64-bit hardware register incremented once per clock cycle, starting from zero at power-up. We can use it to make a timestamp abstraction. As explained by Wikipedia page³⁵², some care is required when using the register for that purpose on modern hardware, but it will suffice to illustrate the idiom implementation. Note that the Singleton pattern is itself somewhat controversial in the OOP community, but that's beyond the scope of this document.

Why use the Singleton pattern in this case? Ordinarily, clients of some ADT will reasonably expect that the states of distinct objects are independent of each other. When using an ADT to represent a single piece of hardware, however, this presumption of independence will not hold because the device is shared by all the objects, unavoidably. The singleton idiom prevents the resulting problems by precluding the existence of multiple objects in the first place.

In this specific case, the time stamp counter hardware is read-only, so the lack of independence is not an issue. Multiple objects would not be a problem. But many devices are not read-only, so the singleton pattern is worth knowing.

First we'll define a singleton ADT representing the TSC register itself, then we will extend that type to add convenience operations for measuring elapsed times. We'll use the design approach of indefinite limited private types without any constructor functions in order to ensure clients cannot create objects of the type. The type will also be tagged for the sake of allowing type extensions. Adding the tagged characteristic doesn't change anything regarding the idiom implementation.

```
with Interfaces;

package Timestamp is

  type Cycle_Counter (<>) is tagged limited private;

  type Cycle_Counter_Reference is access all Cycle_Counter;

  function Counter return not null Cycle_Counter_Reference;

  type Cycle_Count is new Interfaces.Unsigned_64;

  function Sample (This : not null access Cycle_Counter) return Cycle_Count;

private

  type Cycle_Counter is tagged limited null record;

  function Read_TimeStamp_Counter return Cycle_Count with
    Import,
    Convention    => Intrinsic,
    External_Name => "__rdtsc",
    Inline;

  -- This gcc builtin issues the machine instruction to read the time-stamp
  -- counter, i.e., RDTSC, which returns a 64-bit count of the number of
  -- system clock cycles since power-up.
```

(continues on next page)

³⁵¹ https://en.wikipedia.org/wiki/Time_Stamp_Counter

³⁵² https://en.wikipedia.org/wiki/Time_Stamp_Counter

(continued from previous page)

```

function Sample (This : not null access Cycle_Counter)
    return Cycle_Count is
    (Read_TimeStamp_Counter);
    -- The formal parameter This is not referenced

end Timestamp;

```

Note also that the primitive function named Counter is not a constructor — it doesn't return an object of the Cycle_Counter type. As such, it cannot be used as an initial value for a Cycle_Counter object declaration. Clients cannot, therefore, create their own objects of type Cycle_Counter.

Instead, function Counter returns an access value designating an object of the type. Because clients cannot declare objects themselves the function is the only way to get an object, albeit indirectly. Therefore, the function can control how many objects are created. As you will see, the function only creates a single object of the type.

The type Cycle_Counter is completed as a null record because the state is maintained in the hardware register we're reading.

The function Sample reads the timestamp counter register by calling the Read_TimeStamp_Counter function. That second function accesses the TSC register by executing an assembly language instruction dedicated to that purpose. We could have Sample issue that instruction instead, without declaring a separate function, but there is no run-time cost (due to the inlining) and separating them emphasizes that one is a member of the API and the other is an implementation artifact. Note that Sample does not actually reference the formal parameter This. The parameter exists just to make Sample a primitive function. Assuming we don't have a use-clause for Timestamp, to call Sample we could say:

```
TimeStamp.Counter.Sample
```

for example:

```

with Timestamp;
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo_TimeStamp is
begin
    for K in 1 .. 10 loop
        Put_Line (TimeStamp.Counter.Sample'Image);
    end loop;
end Demo_TimeStamp;

```

The above calls the Timestamp.Counter function and then implicitly dereferences the resulting access value to call the Sample function using the distinguished receiver syntax. The resulting number is then converted to a **String** value and output to Standard_Output.

We could have instead used positional call notation for the call to Sample:

```
TimeStamp.Sample (TimeStamp.Counter)
```

In that case we need the package name on the references, or we'd add a use-clause.

The package body is shown below. Only the function Counter has a body because Sample is completed in the package declaration's private part and Read_TimeStamp_Counter is an imported intrinsic, i.e., without a body.

```
package body Timestamp is
```

(continues on next page)

(continued from previous page)

```

The_Instance : Cycle_Counter_Reference;

-----
-- Counter --
-----

function Counter return not null Cycle_Counter_Reference is
begin
    if The_Instance = null then
        The_Instance := new Cycle_Counter;
    end if;
    return The_Instance;
end Counter;

end Timestamp;

```

Function Counter creates the single object that this singleton implementation creates. It does so by lazily allocating an object dynamically. If Counter is never called (because some subclass is used instead) then no object of type Cycle_Counter is created. At most one Cycle_Counter object is ever created.

We could instead declare The_Instance as a Cycle_Counter object in the package body, mark it as aliased, and return a corresponding access value designating it. But when objects are large, declaring one that might never be used is wasteful. The indirection avoids that wasted storage at the cost of an access object, which is small. On the other hand, now the heap is involved.

Note that we could have declared The_Instance in the private part of the package declaration. Type extensions in child packages could then use it, if needed. Presumably we'd make The_Instance be of some access to class-wide type so that extensions could use it to allocate objects of their specific type, otherwise extensions in child packages would have no need for it. But that only saves the storage for an access object in the child packages, so we leave the declaration in the parent package body. See the *Programming by Extension idiom* (page 1199) for a discussion of whether to declare an entity in the package private part or the package body.

Next, we declare a type extension in a child package. The child package body will contain its own object named The_Instance, returning an access value designating the specific extension type. The client API in the package declaration follows that of the parent type Cycle_Counter, but with additional primitives for working with samples.

```

package Timestamp.Sampling is

    type Timestamp_Sampler is new Cycle_Counter with private;

    type Timestamp_Sampler_Reference is access all Timestamp_Sampler;

    function Counter return not null Timestamp_Sampler_Reference with Inline;
    -- returns an access value designating the single instance

    procedure Take_First_Sample (This : not null access Timestamp_Sampler)
        with Inline;
    procedure Take_Second_Sample (This : not null access Timestamp_Sampler)
        with Inline;

    function First_Sample (This : not null access Timestamp_Sampler)
        return Cycle_Count;
    function Second_Sample (This : not null access Timestamp_Sampler)
        return Cycle_Count;
    function Elapsed (This : not null access Timestamp_Sampler)
        return Cycle_Count;

```

(continues on next page)

(continued from previous page)

```

private

  type Timestamp_Sampler is new Cycle_Counter with record
    First : Cycle_Count := 0;
    Second : Cycle_Count := 0;
  end record;

end Timestamp.Sampling;

package body Timestamp.Sampling is

  The_Instance : Timestamp_Sampler_Reference;

  -----
  -- Counter --
  -----

  function Counter return not null Timestamp_Sampler_Reference is
  begin
    if The_Instance = null then
      The_Instance := new Timestamp_Sampler;
    end if;
    return The_Instance;
  end Counter;

  -----
  -- Take_First_Sample --
  -----

  procedure Take_First_Sample (This : not null access Timestamp_Sampler) is
  begin
    This.First := Sample (This);
  end Take_First_Sample;

  -----
  -- Take_Second_Sample --
  -----

  procedure Take_Second_Sample (This : not null access Timestamp_Sampler) is
  begin
    This.Second := Sample (This);
  end Take_Second_Sample;

  -----
  -- First_Sample --
  -----

  function First_Sample (This : not null access Timestamp_Sampler)
    return Cycle_Count is
    (This.First);

  -----
  -- Second_Sample --
  -----

  function Second_Sample (This : not null access Timestamp_Sampler)
    return Cycle_Count is
    (This.Second);

  -----

```

(continues on next page)

(continued from previous page)

```

-- Elapsed --
-----

function Elapsed (This : not null access Timestamp_Sampler)
    return Cycle_Count is
    (This.Second - This.First + 1);

end Timestamp.Sampling;

```

The inherited `Sample` function is called in the two procedures that take the two samples of the timestamp register. The formal parameter `This` is passed to the calls, but as mentioned earlier the argument is not referenced within `Sample`. All the formal parameter does is participate in dispatching the calls to `Sample`, in this case meaning that the inherited version of `Sample` is the one called because `This` is of the extended type.

But `Sample` is not overridden in this child package, therefore effectively we are calling the parent version. Is `Sample` ever likely to be overridden? Arguably not, because it is so directly dependent on the underlying hardware. Of course, some future type extension may override `Sample` for some unforeseen reason — that's the point of making it possible, after all. Presumably the overridden version would also call the parent version, otherwise the timestamp counter would not be accessed. Because we can't say for certain that it will never need to be overridden, we have made `Sample` a primitive function, thus overridable.

Suppose we came to the opposite conclusion, that `Timestamp.Sample` would never need to be overridden. In that case we have some options worth exploring.

Clearly function `Sample` must be part of the client API, but that doesn't force it to be a primitive function.

We could have declared `Sample` in `Timestamp` as a visible non-primitive operation, i.e., without a formal parameter or function result of the ADT type:

```
function Sample return Cycle_Count with Inline;
```

As a non-primitive function it would be neither inherited nor overridable. But we'd still be able to call it in client code.

Yet, as a non-primitive, this version looks like an implementation artifact, hence out of place as part of the visible client API. It isn't illegal by any means, it just *looks* wrong.

Furthermore, if we are going to make `Sample` a non-primitive function, why not remove it and replace it with the other non-primitive function `Read_Timestamp_Counter`? Or make the body of `Sample` call the imported intrinsic, and do away with function `Read_Timestamp_Counter`? There is no clear winner here.

An attractive alternative would be to make `Sample` be a class-wide operation. To do so, we make the formal parameter class-wide instead of removing it:

```
function Sample (This : not null access Cycle_Counter'Class)
    return Cycle_Count
with Inline;
```

In the version above, the formal parameter type is now (anonymous) access to `Cycle_Counter'Class`, i.e., class-wide, so in this version `Sample` can be passed a value designating an object of type `Cycle_Counter` or any type derived from it. We don't want to have a null access value passed so we add that to the parameter specification.

In this version the function is again not a primitive operation and so is neither inherited nor overridable, but because it mentions type `Cycle_Counter` it looks like a reasonable part of an Abstract Data Type. As it happens this version of `Sample` also doesn't actually reference the formal parameter, so it is somewhat unusual. Ordinarily in the body we'd expect the

class-wide formal to be used in dynamic dispatching calls to primitive operations, but that's not required by the language.

Ultimately whether to make `Sample` a primitive operation is a judgment call. We don't know that `Sample` will never need to be overridden so we declare it as a primitive op.

With all that said, here is an example program using the `child` type. Because the timestamp register is updated once per clock cycle, if we know the system clock frequency we can use the counter to measure elapsed time. In the demo below we measure the accuracy of the delay statement by delaying for a known time, with samples taken before and after the delay statement. We can then compare the known delay time to the measured elapsed time, printing the difference.

Note the constant `Machine_Cycles_Per_Second`. Before you run the demo you will likely need to change it in the source code to your machine's clock frequency.

```
with Timestamp.Sampling;    use Timestamp.Sampling;
with Ada.Text_IO;          use Ada.Text_IO;

procedure Demo_Sampling_Cycle_Counter is

  Delay_Interval : constant Duration := 1.0;
  -- arbitrary, change if desired
  Elapsed_Time   : Duration;

  GHz : constant := 1_000_000_000;

  Machine_Cycles_Per_Second : constant := 1.9 * GHz;
  -- This is the system clock rate on the machine running this executable.
  -- It corresponds to the rate at which the time stamp counter hardware is
  -- incremented. Change it according to your target.

  use type Timestamp.Cycle_Count; -- for "<"
begin
  Put_Line ("Using" & Machine_Cycles_Per_Second'Image
    & " Hertz for system clock");

  Put_Line ("Delaying for" & Delay_Interval'Image & " second(s) ...");

  Counter.Take_First_Sample;
  delay Delay_Interval;
  Counter.Take_Second_Sample;

  Put_Line ("First sample           :" & Counter.First_Sample'Image);
  Put_Line ("Second sample          :" & Counter.Second_Sample'Image);

  if Counter.Second_Sample < Counter.First_Sample then
    Put_Line ("RDTSC counter wrapped around!?");
    return;
  end if;

  Elapsed_Time := Duration (Elapsed (Counter)) / Machine_Cycles_Per_Second;

  Put_Line ("Elapsed count           :" & Elapsed (Counter)'Image);
  Put_Line ("Specified delay interval:" & Delay_Interval'Image);
  Put_Line ("Measured delay interval  :" & Elapsed_Time'Image);
end Demo_Sampling_Cycle_Counter;
```

In the above, `Delay_Interval` is set to 1.0 so the program will delay for 1 second, with samples taken from the TSC before and after. Delay statement semantics are such that at least the amount of time requested is delayed, so some value slightly greater than 1 second is expected. There will be overhead too, so an elapsed time slightly larger than requested should be seen. The value of `Delay_Interval` is arbitrary, change it to whatever you like.

If you have set the `Machine_Cycles_Per_Second` properly but still get elapsed measurement values that are much larger than expected or don't make sense at all, it may be that your machine does not support using the `TSC`³⁵³ this way reliably.

35.3 Pros

Ensuring explicit initialization is easily achieved. The abstraction should likely be a private type anyway, and the syntax for the required additional building blocks is light: all are just additional decorations on the declaration of the private type or private extension. The compiler does the rest, either at compile-time itself or via a generated check verified at run-time.

Likewise, ensuring that only the implementation can create objects of a type is straightforward. We take the same approach for ensuring initialization via function calls in object declarations, but then don't provide any such functions. Only the implementation will have the required visibility to create objects of the type, and can limit that number of objects to one (or any other number). Client access to this hidden object must be indirect, but that is not a heavy burden.

35.4 Cons

None.

35.5 Relationship With Other Idioms

The *Abstract Data Type* (page 1187) is assumed, in the form of a private type.

35.6 Notes

Only certain types can have unknown discriminants. For completeness here is the list:

- A private type
- A private extension
- An incomplete type
- A generic formal private type
- A generic formal private type extension
- A generic formal derived type
- Descendants of the above

The types above will either have a corresponding completion or a generic actual parameter to either define the discriminants or specify that there are none.

As we mentioned, `Default_Initial_Condition` is new in Ada 2022. The other implementation, based on indefinite private types, is supported by Ada 2022 but also by earlier versions of the language. However, if the type is also limited, Ada 2005 is the earliest version allowing that implementation. Prior to that version an object of a limited type could not be initialized in the object's declaration.

³⁵³ https://en.wikipedia.org/wiki/Time_Stamp_Counter

TYPE PUNNING

36.1 Motivation

When declaring an object, the type chosen by the developer is presumably one that meets the operational requirements. Sometimes, however, the chosen type is not sufficient for clients of that object. Normally that situation would indicate a design error, but not necessarily.

Consider a device driver that receives external data, such as a network or serial I/O driver. Typically the driver presents incoming data to clients as arrays of raw bytes. That's how the data enter the receiving machine, so that's the most natural representation at the lowest level of the device driver code. However, that representation is not likely sufficient for the higher-level clients. These clients are not necessarily at the application level, they may be lower than that, but they are clients because they use the data presented.

For example, the next level of the network processing code must interpret the byte arrays in order to implement the network protocol. Interpreting the data requires reading headers that are logically contained within slices of those bytes. These headers are naturally represented as record types containing multiple fields. How, then, can the developer apply such record types? An array of bytes contains bytes, not headers.

Stated generally, on occasion a developer needs to manipulate or access the value of a given object in a manner not supported by the object's type. The issue is that the compiler will enforce the type model defined by the language, to some degree of rigor, potentially resulting in the rejection of the alternative access and manipulation code. In such cases the developer must circumvent this enforcement. That's the purpose of this idiom.

A common circumvention technique, across programming languages, is to apply multiple distinct types to a single given object. Doing so makes available additional operations or accesses not provided by the type used to declare the object in the first place. The technique is known as **type punning**³⁵⁴ in the programming community because different types are used for the same object in much the same way that a pun in natural languages uses different meanings for words that sound the same when spoken.

Ada is accurately described as "a language with inherently reliable features, only compromised via explicit escape hatches having well-defined and portable semantics when used appropriately."³⁵⁷ The foundation for this reliability is static named typing with rigorous enforcement by the compiler.

Specifically, the Ada compiler checks that the operations and values applied to an object are consistent with the one type specified when the object is declared. Any usage not consistent with that type is rejected by the compiler. Similarly, the Ada compiler also checks that any type conversions involve only those types that the language defines as meaningfully convertible.

³⁵⁴ https://en.wikipedia.org/wiki/Type_punning

³⁵⁷ S. Tucker Taft. *Post in Internet Relay Chat on Comp.Lang.Ada channel.* https://groups.google.com/g/comp.lang.ada/c/9WXgvv8Xjuw/m/JMyo9_P7nxAJ, 1993.

By design, this strong typing model does not lend itself to circumvention (thankfully). That's the point of Ada's *escape hatches* — they provide standard ways to circumvent these and other checks. To maintain the integrity of the type model not many escape hatches exist. The most commonly used of these, *unchecked conversion*, allows type conversion between arbitrary types. Unchecked conversions remain explicit, but the compiler does not limit them to the types defined as reasonable by the language.

36.2 Implementation(s)

There are two common approaches for expressing type punning in Ada. We show both in the following subsections. The purpose in both approaches is to apply a different type, thereby making available a different type-specific view of the storage.

36.2.1 Overlays

The first approach applies an alternative type to an existing object by declaring another object at the same location in memory but with a different type. Given an existing object, the developer:

1. declares (or reuses) an alternative type that provides the required operations and values not provided by the existing object's type, and
2. declares another object of this alternative type, and
3. as part of the new object's declaration, specifies that this new object shares some or all of the storage occupied by the existing object.

The result is a partial or complete storage overlay. Because there are now multiple distinct types involved, there are multiple views of that shared storage, each view providing different operations and values. Thus, the shared storage can be legally manipulated in distinct ways. As usual, the Ada compiler verifies that the usage corresponds to the type view presented by the object name referenced.

For example, let's say that we have an existing object, and that a signed integer is most appropriate for its type. On some occasions let's also say we need to access individual bits within that existing object. Signed integer types don't support bit-level operations, unlike unsigned integers, but we've said that a signed type is the best fit for the bulk of the usage.

One of the ways to enable bit access, then, is to apply another type that does have bit-level operations. We could overlay the existing object with an unsigned integer type of the same size, but let's take a different approach for the sake of illustration. Instead, we'll declare an array type with components that can be represented as a single bit. The length of the array type will reflect the number of bits used by the signed integer type so that the entire object will be overlaid. (A record type would work too, with a component allocated to each bit.)

The type **Boolean** will suffice for the array component type as long as we force the single-bit representation. **Boolean** array components are likely to be represented as individual bytes otherwise. Alternatively, we could just make up an integer type with range `0 .. 1` but that seems unnecessary, unless numeric values would make the code clearer. (Maybe the requirements specify ones and zeros for the bit values.) In any case we'll need to force single bits for the components.

Assuming values of type **Integer** require exactly 32 bits, the following code illustrates the approach:

```
type Bits32 is array (0 .. 31) of Boolean with Component_Size => 1;

X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

In the above, we use `X'Address` to query the starting address of object `X`. We use that address to specify the location of the overlay object `Y`. As a result, `X` and `Y` start at the same address.

We marked `X` as explicitly **aliased** because **Integer** is not a by-reference type. The **Address** attribute is not required to provide a useful result otherwise. (Maybe the compiler would have put `X` into a register, for example.) The compiler, seeing `'Address` applied, would probably do the right thing anyway, but this makes it certain.

Here is a simple main program that illustrates the approach.

Listing 1747: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5     X : aliased Integer := 42;
6
7     type Bits32 is array (0 .. 31) of Boolean
8         with Component_Size => 1;
9
10    Y : Bits32 with Address => X'Address;
11
12 begin
13     X := Integer'First;
14     Put_Line (X'Image);
15     for Bit in Bits32'Range loop
16         Put (if Y (Bit) then '1' else '0');
17     end loop;
18     New_Line;
19 end Main;
```

Object `Y` starts at the same address as `X` and has the same size, so all references to `X` and `Y` refer to the same storage. The source code can manipulate that memory as either a signed integer or as an array of bits, including individual bit access, using the two object names. The compiler will ensure that every reference via `X` is compatible with the integer view, and every reference via `Y` is compatible with the array view.

In the above example, we've ignored the endianness issue. If you wanted to change the sign bit, for example, or display the bits in the "correct" order, you'd need to handle that detail.

This expression of type-punning does not use an escape hatch but does achieve the effect. (We don't include address clauses as an escape hatch because address clauses aren't dedicated to overlaying multiple objects of different types. On the other hand, even one Ada object with an address specified overlays that object's view with the machine storage view of that address...)

36.2.2 Unchecked Conversions on Address Values

The common implementation of type punning, across multiple languages, involves converting the address of a given object into a pointer designating the alternative type to be applied. Dereferencing the resulting pointer provides a different compile-time view of the object. Thus, the operations defined by the alternative type are made applicable to the object.

Expressing this approach in Ada requires unchecked conversion because, in Ada, address values are semantically distinct from pointer values (*access values*). An access value might be represented by an address value, but because architectures vary, that implementation is not guaranteed. Therefore, the language does not define checked conversions between addresses and access values. We need the escape hatch.

Unchecked conversion requires instantiation of the generic function `Ada.Unchecked_Conversion`, including a context clause for that unit, making it a relatively heavy mechanism. This heaviness is intentional, and the `with_clause` at the top of the client unit makes it noticeable. Although ubiquitous use strongly suggests abuse of the type model, in this case unchecked conversion is necessary. Nevertheless, we'd hide its use within the body of some visible unit.

Let's start with a simple example. There is an accelerometer that provides three signed 16-bit acceleration values, one for each axis. Accelerations can be both positive and negative so the signed type is appropriate. These values are read from the device as two unsigned bytes. The two bytes are read individually so two reads are required per acceleration value. (This is an actual, real-world device.) Because the acceleration values are signed 16-bit integers, we need to convert two unsigned bytes into a single signed 16-bit quantity. We can use type punning, based on a pointer designating the 16-bit signed type, to achieve that effect. There are certainly other ways to do this, but we're starting with something simple for the sake of illustrating this idiom.

In the following fragment, type `Acceleration` is a signed 16-bit integer type already declared elsewhere:

```
type Acceleration_Pointer is access all Acceleration
  with Storage_Size => 0;

function As_Acceleration_Pointer is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => Acceleration_Pointer);
```

The access type is general, not pool-specific, but that is optional. We tell the compiler to reserve no storage for the access type because an allocation would be an error that we want the compiler to catch. Whether or not the compiler actually reserves storage for an individual access type is implementation-dependent, but this way we can be sure. In any case the compiler will reject any allocations.

Given this access type declaration we can then instantiate `Ada.Unchecked_Conversion`. The resulting function name is a matter of style but is appropriate because the function allows us to treat an address as a pointer to an `Acceleration` value. We aren't changing the address value, we're only providing another view of that value, which is why the function name is not `To_Acceleration_Pointer`.

The following is the device driver routine for getting the scaled accelerations from the device. The type `Three_Axis_Accelerometer` is the device driver *ADT* (page 1187), and type `Axes_Accelerations` is a record type containing the three axis values. The procedure gets the raw acceleration values from the device and scales them per the current device sensitivity, returning all three in the mode-out record parameter.

```
procedure Get_Accelerations
  (This : Three_Axis_Accelerometer;
   Axes : out Axes_Accelerations)
is
  Buffer : array (0 .. 5) of UInt8 with Alignment => 2, Size => 48;
  Scaled : Float;

  type Acceleration_Pointer is access all Acceleration
    with Storage_Size => 0;

  function As_Acceleration_Pointer is new Ada.Unchecked_Conversion
    (Source => System.Address, Target => Acceleration_Pointer);

begin
  This.Loc_IO_Read (Buffer (0), OUT_X_L);
  This.Loc_IO_Read (Buffer (1), OUT_X_H);
  This.Loc_IO_Read (Buffer (2), OUT_Y_L);
```

(continues on next page)

(continued from previous page)

```

This.Loc_IO_Read (Buffer (3), OUT_Y_H);
This.Loc_IO_Read (Buffer (4), OUT_Z_L);
This.Loc_IO_Read (Buffer (5), OUT_Z_H);

Get_X : declare
  Raw : Acceleration renames
    As_Acceleration_Pointer (Buffer (0)'Address).all;
begin
  Scaled := Float (Raw) * This.Sensitivity;
  Axes.X := Acceleration (Scaled);
end Get_X;

Get_Y : declare
  Raw : Acceleration renames
    As_Acceleration_Pointer (Buffer (2)'Address).all;
begin
  Scaled := Float (Raw) * This.Sensitivity;
  Axes.Y := Acceleration (Scaled);
end Get_Y;

Get_Z : declare
  Raw : Acceleration renames
    As_Acceleration_Pointer (Buffer (4)'Address).all;
begin
  Scaled := Float (Raw) * This.Sensitivity;
  Axes.Z := Acceleration (Scaled);
end Get_Z;
end Get_Accelerations;

```

This procedure first reads the six bytes representing all three acceleration values into the array `Buffer`. Procedure `Loc_IO_Read` is defined by the driver [ADT](#) (page 1187). The constants `OUT_n_L` and `OUT_n_H`, also defined by the driver, specify the low-order and high-order bytes requested for the given n axis. Then the `declare` blocks do the actual scaling and that's where the type punning is applied to the `Buffer` content.

In each block, the address of one of the bytes in the array is converted into an access value designating a two-byte `Acceleration` value. The X acceleration is first in the buffer, so the address of `Buffer (0)` is converted. Likewise, the address of `Buffer (2)` is converted for the Y axis value, and for the Z value, `Buffer (4)` is converted. (We could have said `Buffer 'Address` instead of `Buffer (0)'Address`, they mean the same thing, but an explicit index seemed more clear, given the need for the other indexes.)

But we want the designated axis acceleration value, not the access value, so we also dereference the converted access value via `.all`, and rename the result for convenience. The name is `Raw` because the value needs to be scaled. Each dereference reads two bytes, i.e., the bytes at indexes 0 and 1, or 2 and 3, or 4 and 5.

That's the way the device driver is written currently, but it could be simpler. Clients always get all three accelerations via this procedure, so we could have used unchecked conversion to directly convert the entire array of six bytes into a value of the record type `Axes_Accelerations` containing the three 16-bit components. Type punning would not be required in that case. (The components would still need scaling, of course.)

Note that to get individual values we can't just convert a slice of the array because that's illegal: array slices cannot be converted. We'd need some other way to refer to a two-byte pair within the array. Type punning would be an appropriate approach.

For that matter we could use type punning but have the record type be the designated type returned from the address conversion, rather than a single axis value. Then we'd just convert `Buffer 'Address` and not need to specify array indexes as `all`. This would be the same as converting the array to the record type, but with a level of indirection added.

For the network packet example, we want to apply record type views to arbitrary sequences within an array of raw bytes, so indexing will be required. Just as we indexed into the accelerometer Buffer for the addresses of the individual 16-bit acceleration values, we can index into the network packet array to get the starting addresses of the individual headers. Regular record component access syntax can then be used. Reading the record components reads the corresponding raw bytes in the array.

For a specific example, we can read the IP header in a packet's array of bytes using the header's record type and an access type designating that record type:

```
Min_IP_Header_Length : constant := 20;

-- IP packet header RFC 791.
type IP_Header is record
  Version      : UInt4;
  Word_Count    : UInt4;
  Type_of_Service : UInt8;
  Total_Length  : UInt16;
  Identifier    : UInt16;
  Flags_Offset  : UInt16;
  Time_To_Live  : UInt8;
  Protocol      : Transport_Protocol;
  Checksum      : UInt16;
  Source        : IP_Address;
  Destination   : IP_Address;
end record with
  Alignment => 2,
  Size      => Min_IP_Header_Length * 8;

for IP_Header use record
  Version      at 0 range 4 .. 7;
  Word_Count    at 0 range 0 .. 3;
  Type_of_Service at 1 range 0 .. 7;
  Total_Length  at 2 range 0 .. 15;
  Identifier    at 4 range 0 .. 15;
  Flags_Offset  at 6 range 0 .. 15;
  Time_To_Live  at 8 range 0 .. 7;
  Protocol      at 9 range 0 .. 7;
  Checksum      at 10 range 0 .. 15;
  Source        at 12 range 0 .. 31;
  Destination   at 16 range 0 .. 31;
end record;
for IP_Header'Bit_Order use System.Low_Order_First;
for IP_Header'Scalar_Storage_Order use System.Low_Order_First;

type IP_Header_Access is access all IP_Header;
pragma No_Strict_Aliasing (IP_Header_Access);

-- and so on, for the other kinds of headers...
```

Note that `pragma No_Strict_Aliasing` stops the compiler from doing some optimizations, based on the assumption of a lack of aliasing, that could cause unexpected results in this approach.

```
function As_IP_Header_Access is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => IP_Header_Access);

function As_ARP_Header_Access is new Ada.Unchecked_Conversion
  (Source => System.Address, Target => ARP_Packet_Access);

-- and so on, for the other kinds of headers...
```

We can then implement a function, visible to clients, for acquiring the access value from a

given memory buffer's data:

```
function IP_Hdr (This : Memory_Buffer) return IP_Header_Access is
  (As_IP_Header_Access (This.Packet.Data (IP_Pos)'Address));
```

In the function, `Data` is the packet's array of raw bytes, and `IP_Pos` is a constant specifying the index into the array corresponding to the first byte of the IP header. As you can see, this is the same approach as shown earlier for working with an array of bytes containing acceleration values.

Similar functions support [ARP³⁵⁵](#) headers, [TCP³⁵⁶](#) headers, and so on.

36.3 Pros

Both approaches work and are fairly simple, although the first is simplest. The second approach, based on converting addresses to access values, is more flexible. That's because the address to be converted can be changed at run-time, whereas the object overlay specifies the address exactly once during elaboration.

36.4 Cons

We're assuming access values are represented as addresses. There's no guarantee of that. But on typical architectures it will likely work.

That said, not all types can support the address conversion approach. In particular, unconstrained array types may not work correctly because of the existence of the additional in-memory representation of the bounds. An access value designating such an object might point at the bounds of the array whereas the address of the object would point to the first element.

In either approach, the developer is responsible for the correctness of the address values applied, either for the second object's declaration or for the pointer conversion. For example, this includes the alternative type's alignment. Otherwise, all bets are off.

36.5 Relationship With Other Idioms

None.

36.6 Notes

Generic package `System.Address_To_Access_Conversions` is an obvious alternative to our use of unchecked conversions between addresses and access values. The generic is convenient: it provides the access type as well as functions for converting in both directions. But it will require an instantiation for each designated type, so it offers no reduction in the number of instantiations required over that of `Ada.Unchecked_Conversion`. (For more details on this generic package, please refer to the section on [access and address](#) (page 993).)

Moreover, because that generic package is defined by the language, the naïve user might think it will work for all types. It might not. Unconstrained array types remain a potential problem. For that reason, the GNAT implementation issues a warning in such cases.

³⁵⁵ https://en.wikipedia.org/wiki/Address_Resolution_Protocol

³⁵⁶ https://en.wikipedia.org/wiki/Transmission_Control_Protocol

USING BUILDING BLOCKS TO EXPRESS INHERITANCE IDIOMS

37.1 Motivation

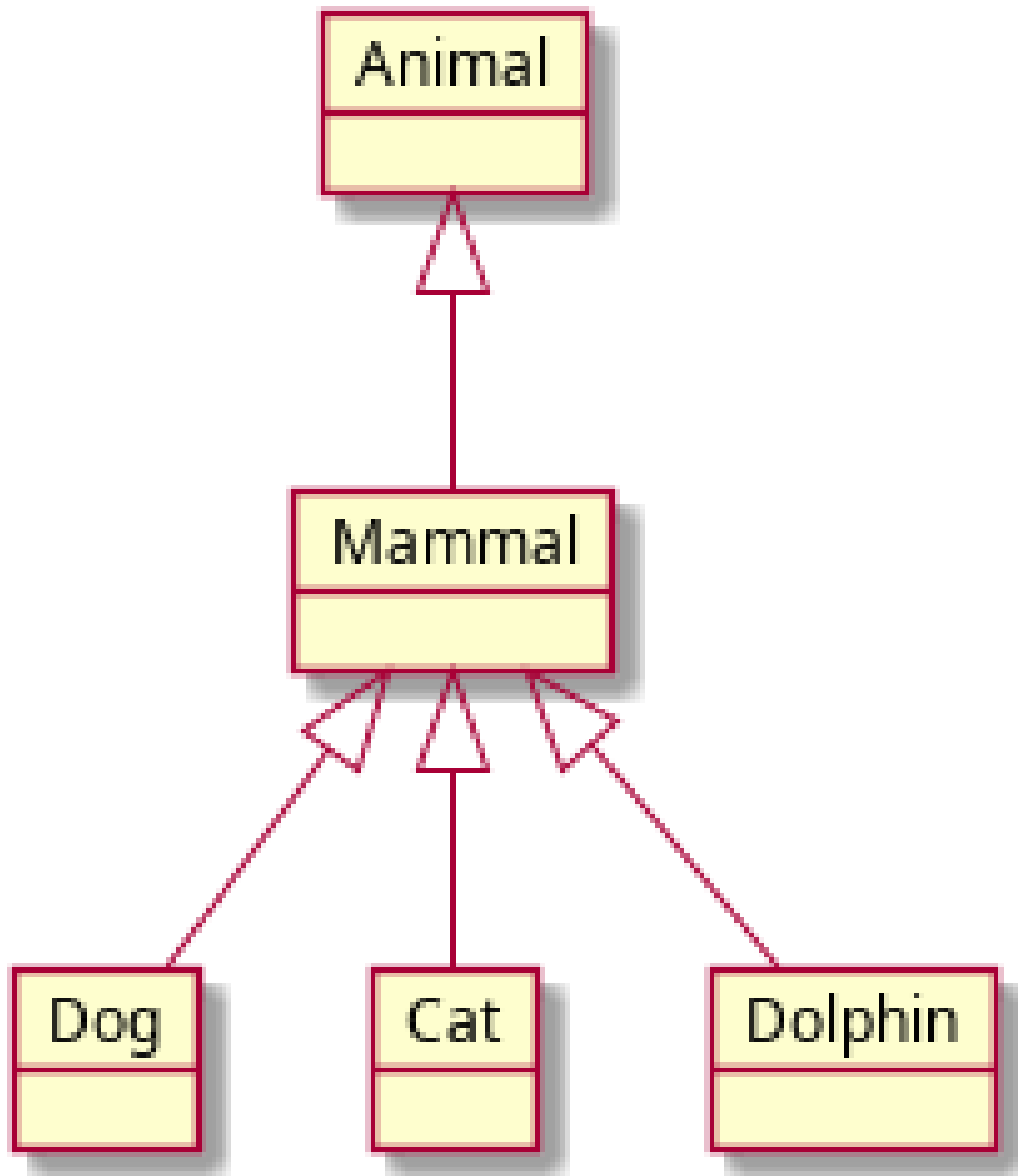
Betrand Meyer's magisterial book on OOP³⁶⁰ includes a taxonomy of inheritance idioms. Two especially well-known entries in that taxonomy are [Subtype Inheritance](#)³⁵⁸ and [Implementation Inheritance](#)³⁵⁹. The name of the first idiom is perhaps confusing from an Ada point of view because Ada subtypes have a different meaning. In Ada terms we are talking about *derived types*. A derived type is a new, distinct type based on (i.e., derived from) some existing type. We will informally refer to the existing ancestor type as the *parent* type, and the new type as the *child* type. The term *Subtype* in the idiom name refers to the child type.

Subtype Inheritance is the most well-known idiom for inheritance because it's based on the notion of a taxonomy, in which categories and disjoint subcategories are identified. For example, we can say that dogs, cats, and dolphins are mammals, and that all mammals are animals:

³⁶⁰ B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

³⁵⁸ <https://en.wikipedia.org/wiki/Subtyping>

³⁵⁹ [https://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))



By saying that the subcategories are disjoint we mean that, for example, dogs are neither cats nor dolphins and cannot be treated as if they are.

In software, we use various constructs to represent the categories and subcategories and use inheritance to organize them. As mentioned above, in Ada, we express that inheritance via derived types representing the categories and subcategories. Ada's strong typing ensures they are treated as disjoint entities.

Although the derived child type is distinct from the parent type, the child is the same *kind* as the parent type. Some authors use *kind of* as the name for the relationship between the child and parent. Meyer uses the term *is-a* [Page 1237, 360](#), a popular term that we will use too. For example, a cat *is a* mammal, and also is an animal.

The fundamental difference between *Subtype Inheritance* (page 1241) and *Implementation Inheritance* (page 1244) is whether clients have compile-time visibility to the *is-a* relationship between the parent and child types. The relationship exists in both idioms but is only

visible to clients in one. In Subtype Inheritance, clients do have compile-time visibility to the relationship, while in Implementation Inheritance, clients don't have that visibility.

Consequently, with Subtype Inheritance, all of the inherited operations become part of the child type's visible interface. In contrast, with Implementation Inheritance, none of those parent capabilities are part of the visible interface: the inherited parent capabilities are only available internally, to implement the child type's representation and its primitive operations.

37.1.1 Building Blocks

Ada uses distinct *building block* constructs to compose types that have specific characteristics and capabilities. In particular, Ada packages, with their control over compile-time visibility, are modules. Private types are combined with packages to define *abstract data types* (page 1187) having hidden representations. Sets of related types are presented explicitly by class-wide types.

In addition, simple reserved words may be attached to a type declaration to refine or expand the capabilities of the type. These type declarations include declarations for derived types, providing considerable flexibility and expressive power for controlling the client's view of the child and parent types.

For example, in Ada, full dynamic OOP capabilities require type declarations to be decorated with the reserved word **tagged**. However, from its earliest days, Ada has also supported a static form of inheritance, using types that are not tagged. The implementation we describe below works with both forms of inheritance.

The developer also has a choice of whether the parent type and/or the child type is a private type. Using private types is the default design choice, for the sake of designing in terms of abstract data types, but is nevertheless optional.

In addition, a type can be both private and tagged. This possibility raises the question of whether the type is *visibly tagged*, i.e., whether the client view of the type includes the tagged characteristic, and hence the corresponding capabilities. Recall that a private type is declared in two steps: the first part occurs in the visible part of the package and introduces the type name to clients. The second part — the type completion — appears in the package private part and specifies the type's actual representation. The question arises because the first step, i.e., the declaration in the package's visible part, need not be tagged, yet can be tagged in the completion in the package private part. For example:

```
package P is
  type Foo is private; -- not visibly tagged for clients
  -- operations on type Foo
private
  type Foo is tagged record -- tagged completion
    ...
  end record;
end P;
```

In the above, Foo is not visibly tagged except in the package private part and the package body. As a consequence, the capabilities of tagged types are not available to clients using type Foo. Clients cannot refer to Foo's *Class*, for example. (The opposite arrangement — tagged in the visible client view but not actually tagged in the private view — is not legal, because clients would be promised capabilities that are not actually available.)

When the parent type is tagged, the type derivation syntax for the child is a *type extension* declaration that introduces the child type's name, specifies the parent type, and then extends the parent representation with child-specific record components, if any. For example:

```
type Child is new Parent with record ... end record;
```

Even though the child type declaration does not include the reserved word **tagged** the child will be a tagged type because the parent type is tagged. The compiler would not allow the extension construct for a non-tagged parent type.

Just as a private type can be visibly tagged or not, a private type can be *visibly derived* or not. When it is visibly derived, clients have a view of the private type that includes the fact of the derivation from the parent type. Otherwise, clients have no view of the parent type. Whether or not the child is visibly derived, the representation is not compile-time visible to clients, as for any private type. For example, type `Foo` is not visibly derived in the following:

```
package P is
  type Foo is tagged private; -- visibly tagged but not visibly derived
  -- ...
end P;
```

To be visibly derived, we declare the child type as a private type using a *private extension*. A private extension is like a type extension, in that it introduces the child type name and the parent type. But like any private type declaration, it does not specify the type's representation. This is the first of the two steps for declaring a private type; hence it appears in the package visible part. For example:

```
with ...
package P is
  type Child is new Parent with private; -- visibly derived from Parent
private
  type Child is new Parent with record ... end record;
end P;
```

The representation additions are not expressed until the private type's completion in the package private part, using a type extension. The steps are the same two for any private type: a declaration in the package visible part, with a completion in the package private part. The difference is the client visibility to the parent type.

37.2 Implementation(s)

There are two implementations presented, one for each of the two inheritance idioms under discussion. First, we will specify our building block choices, then show the two idiom expressions in separate subsections.

- We use tagged types for the sake of providing full OOP capabilities. That is the most common choice when inheritance is involved. The static form of inheritance has cases in which it is useful, but those cases are very narrow in applicability.
- We assume that the parent type and the child type are both private types, i.e., abstract data types, because that is the best practice. See the [Abstract Data Type idiom](#) (page 1187) for justification and details.
- To provide the most general capabilities, we assume the parent type is visibly tagged.
- We're going to declare the child type in a distinct, dedicated package, following the [ADT idiom](#) (page 1187). This package may or may not be a child of the parent package. This implementation's approach does not require a child package's special compile-time visibility, although a child package is often necessary for the sake of that visibility.
- Whether the child type is visibly derived will vary with the [inheritance idiom](#) (page 1244) implementation.

To avoid unnecessary code duplication, we use the same parent type, declared as a simple tagged private type, in the examples for the two idiom implementations. The parent type could itself be derived from some other tagged type, but that changes nothing conceptually significant. We declare parent type in package `P` as follows:

```

package P is
  type Parent is tagged private; -- visibly tagged
  -- primitive operations with type Parent as the
  -- controlling formal parameter
private
  type Parent is tagged record ... end record;
end P;

```

37.2.1 Subtype Inheritance

Recall that Subtype Inheritance requires clients to have compile-time visibility to the *is-a* relationship between the child and parent types. We can satisfy that requirement if we make the child visibly derived from the parent. Hence we declare the private type as a private extension in the visible part of the package:

```

with P; use P;
package Q is
  type Child is new Parent with private;
  -- implicit, inherited primitive Parent operations declared here,
  -- now for type Child
  -- additional primitives for Child explicitly declared, if any
private
  type Child is new Parent with record ... end record;
end Q;

```

The primitive operations from the parent type are implicitly declared immediately after the private extension declaration. That means those operations are in the visible part of the package, hence clients can invoke them. Any additional operations for the client interface will be explicitly declared in the visible part as well, as will any overriding declarations for those inherited operations that are to be changed.

For example, here is a basic bank account *ADT* (page 1187) that we will use as the parent type in a derivation:

Listing 1748: bank.ads

```

1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2 with Ada.Containers.Doubly_Linked_Lists;
3
4 package Bank is
5
6   type Basic_Account is tagged private
7     with Type_Invariant'Class => Consistent_Balance (Basic_Account);
8
9   function Consistent_Balance (This : Basic_Account) return Boolean;
10
11   type Currency is delta 0.01 digits 12;
12
13   procedure Deposit (This : in out Basic_Account;
14                     Amount : Currency) with
15     Pre'Class => Open (This) and Amount > 0.0,
16     Post'Class => Balance (This) = Balance (This)'Old + Amount;
17
18   procedure Withdraw (This : in out Basic_Account;
19                     Amount : Currency) with
20     Pre'Class => Open (This) and Funds_Available (This, Amount),
21     Post'Class => Balance (This) = Balance (This)'Old - Amount;
22
23   function Balance (This : Basic_Account) return Currency
24     with Pre'Class => Open (This);

```

(continues on next page)

(continued from previous page)

```

25
26 procedure Report_Transactions (This : Basic_Account)
27   with Pre'Class => Open (This);
28
29 procedure Report (This : Basic_Account)
30   with Pre'Class => Open (This);
31
32 function Open (This : Basic_Account) return Boolean;
33
34 procedure Open
35   (This          : in out Basic_Account;
36    Name          : String;
37    Initial_Deposit : Currency)
38   with Pre'Class => not Open (This),
39        Post'Class => Open (This);
40
41 function Funds_Available (This : Basic_Account;
42                           Amount : Currency) return Boolean is
43   (Amount > 0.0 and then Balance (This) >= Amount)
44   with Pre'Class => Open (This);
45
46 private
47
48   package Transactions is new
49     Ada.Containers.Doubly_Linked_Lists (Element_Type => Currency);
50
51   type Basic_Account is tagged record
52     Owner          : Unbounded_String;
53     Current_Balance : Currency := 0.0;
54     Withdrawals     : Transactions.List;
55     Deposits        : Transactions.List;
56   end record;
57
58   function Total (This : Transactions.List) return Currency is
59     (This'Reduce ("+", 0.0));
60
61 end Bank;

```

We could then declare an interest-bearing bank account using Subtype Inheritance:

Listing 1749: bank-interest_bearing.ads

```

1 package Bank.Interest_Bearing is
2
3   type Account is new Basic_Account with private;
4
5   overriding
6   function Consistent_Balance (This : Account) return Boolean;
7
8   function Minimum_Balance (This : Account) return Currency;
9
10  overriding
11  procedure Open
12    (This          : in out Account;
13     Name          : String;
14     Initial_Deposit : Currency)
15    with Pre => Initial_Deposit >= Minimum_Balance (This);
16
17  overriding
18  procedure Withdraw (This : in out Account; Amount : Currency);
19

```

(continues on next page)

(continued from previous page)

```

20  function Penalties_Accrued (This : Account) return Currency;
21  function Interest_Accrued (This : Account) return Currency;
22
23  private
24
25  type Account is new Basic_Account with record
26      Penalties      : Transactions.List;
27      Interest_Earned : Transactions.List;
28      Days_Under_Minimum : Natural := 0;
29  end record;
30
31  end Bank.Interest_Bearing;

```

The new type `Bank.Interest_Bearing.Account` inherits all the `Basic_Account` operations in the package visible part. They are, therefore, available to clients. Some of those inherited operations are overridden so that their behavior can be changed. Additional operations specific to the new type are also declared in the visible part so they are added to the client API.

The package private part and the body of package `Bank.Interest_Bearing` have visibility to the private part of package `Bank` because the new package is a child of package `Bank`. That makes the private function `Bank.Total` visible in the child package, along with the components of the record type `Basic_Account`.

Note that there is no language requirement that the actual parent type in the private type's completion be the one named in the private extension declaration presented to clients. The parent type in the completion must only be in the same derivation class — be the same kind of type — so that it satisfies the *is-a* relationship stated to clients.

For example, we could start with a basic graphics shape:

```

package Graphics is
  type Shape is tagged private;
  -- operations for type Shape ...
  ...
end Graphics;

```

We could then declare a subcategory of `Shape` that allows translation in some 2-D space:

```

package Graphics.Translatable is
  type Translatable_Shape is new Graphics.Shape with private;
  procedure Translate (This : in out Translatable_Shape; X, Y : in Float);
  ...
end Graphics.Translatable;

```

Given that, we could now declare another type visibly derived from `Shape`, but using `Translatable_Shape` as the actual parent type:

```

with Graphics;
private with Graphics.Translatable;
package Geometry is
  type Circle is new Graphics.Shape with private;
  -- operations for type Circle, inherited from Shape,
  -- and any new ops added ...
private
  use Graphics.Translatable;
  type Circle is new Translatable_Shape with record ... end record;
end Geometry;

```

In the type extension that completes type `Circle` in the package private part above, the extended parent type is not the one presented to clients, i.e., `Graphics.Shape`. Instead, the parent type is another type that is derived from type `Shape`. That substitution is legal

and reasonable because `Translatable_Shape` necessarily can do anything that `Shape` can do. To understand why that is legal, it is helpful to imagine that there is a *contract* between the package public part and the private part regarding type `Circle`. As long as `Circle` can do everything promised to clients — i.e., inherited `Shape` operations — the contract is fulfilled. `Circle` inherits `Shape` operations because `Translatable_Shape` inherits those operations. The fact that `Circle` can do more than is contractually required by the client view is perfectly fine.

37.2.2 Implementation Inheritance

Recall that with Implementation Inheritance clients do not have compile-time visibility to the *is-a* relationship between the parent and child types. We meet that requirement by not making the child visibly derived from the parent. Therefore, we declare the child type as a simple tagged private type and only mention the parent in the child type's completion in the package private part:

```
with P; use P;
package Q is
  type Child is tagged private;
  -- explicitly declared primitives for Child
private
  type Child is new Parent with record ...
  -- implicit, inherited primitive operations with type Child
  -- as the controlling formal parameter
end Q;
```

The primitive operations from the parent type are implicitly declared immediately after the type extension, but these declarations are now located in the package private part. Therefore, the inherited primitive operations are not compile-time visible to clients. Hence clients cannot invoke them. These operations are only visible (after the type completion) in the package private part and the package body, for use with the implementation of the explicitly declared primitive operations.

For example, we might use a *controlled type* in the implementation of a tagged private type. These types have procedures `Initialize` and `Finalize` defined as primitive operations. Both are called automatically by the compiler. Clients generally don't have any business directly calling them so we usually use implementation inheritance with controlled types. But if clients did have the need to call them we would use Subtype Inheritance instead, to make them visible to clients.

For example, the following is a generic package providing an abstract data type for unbounded queues. As such, the `Queue` type uses dynamic allocation internally. This specific version automatically reclaims the allocated storage when objects of the `Queue` type cease to exist:

Listing 1750: `unbounded_sequential_queues.ads`

```
1 with Ada.Finalization;
2 generic
3   type Element is private;
4 package Unbounded_Sequential_Queues is
5
6   type Queue is tagged limited private;
7
8   procedure Insert (Into : in out Queue; Item : Element) with
9     Post => not Empty (Into) and
10     Extent (Into) = Extent (Into)'Old + 1;
11   -- may propagate Storage_Error
12
13   procedure Remove (From : in out Queue; Item : out Element) with
```

(continues on next page)

(continued from previous page)

```

14     Pre => not Empty (From),
15     Post => Extent (From) = Natural'Max (0, Extent (From)'Old - 1);
16
17     procedure Reset (This : in out Queue) with
18         Post => Empty (This) and Extent (This) = 0;
19
20     function Extent (This : Queue) return Natural;
21
22     function Empty (This : Queue) return Boolean;
23
24 private
25
26     type Node;
27
28     type Link is access Node;
29
30     type Node is record
31         Data : Element;
32         Next : Link;
33     end record;
34
35     type Queue is new Ada.Finalization.Limited_Controlled with
36         record
37             Count : Natural := 0;
38             Rear  : Link;
39             Front : Link;
40         end record;
41
42     overriding procedure Finalize (This : in out Queue) renames Reset;
43
44 end Unbounded_Sequential_Queues;

```

The basic operation of assignment usually does not make sense for an abstraction represented as a linked list, so we declare the private type as *limited*, in addition to tagged and private, and then use the language-defined limited controlled type for the type extension completion in the private part.

Procedures Initialize and Finalize are inherited immediately after the type extension. Both are null procedures that do nothing. We can leave Initialize as-is because initialization is already accomplished via the default values for the Queue components. On the other hand, we want finalization to reclaim all allocated storage so we cannot leave Finalize as a null procedure. By overriding the procedure, we can change the implementation. That change is usually accomplished by placing the corresponding procedure body in the package body. However, in this case we have an existing procedure named Reset that is part of the visible (client) API. Reset does exactly what we want Finalize to do, so we implement the overridden Finalize by saying that it is just another name for Reset. No completion body for Finalize is then required or allowed. This approach has the same semantics as if we explicitly wrote a body for Finalize that simply called Reset, but this is more succinct. Clients can call Reset whenever they want, but the procedure will also be called automatically, via Finalize, when any Queue object ceases to exist.

37.3 Pros

The two idioms are easily composed simply by controlling where in the enclosing package the parent type is mentioned: either in the declaration of the private child type in the package visible part or in the child type's completion in the package private type.

37.4 Cons

Although the inheritance expressions are simple by themselves, the many ancillary design choices can make the design effort seem more complicated than it really is.

37.5 Relationship With Other Idioms

We assume the *Abstract Data Type idiom* (page 1187), so we are using private types throughout. That includes the child type, and, as we saw, allows us to control the compile-time visibility to the parent type.

PROVIDING COMPONENT ACCESS TO ENCLOSING RECORD OBJECTS

38.1 Motivation

In some design situations we want to have a record component that is of a task or protected type. That in itself is trivially accomplished because task types and protected types can be used to declare record components. But there's more to this idiom.

We would want a task type or protected type record component when:

- a) a task or protected object (PO) is required to implement part — but not all — of the record type's functionality, and
- b) each such task or PO is intended to implement its functionality only for the object logically containing that specific task object or protected object. The record object and contained task/PO object pair is a functional unit, independent of all other such units.

This idiom applies to both enclosed task types and protected types, but for simplicity let's assume the record component will be of a protected type.

As part of a functional unit, the PO component will almost certainly be required to reference the other record components in the enclosing record object. That reference will allow the PO to read and/or update those other components. Note that these record components include discriminants, if any.

To be a functional unit, the record object referenced by a given PO in this relationship must be the same record object at run-time that contains that specific PO instance. That will allow the PO instance to implement the functionality for the specific record object containing that PO instance.

Unless we arrange it, that back-reference from the protected object to the record object isn't provided. Consider the following:

```
package Q is
  protected type P is ... end P;
  type R is record
    ...
    Y : P;
  end record;
end Q;
```

During execution, whenever an object of type `Q.R` is declared or allocated, at run-time we will have two objects, instances of two distinct types — the record object and the protected object. Let's say that a client declares an object `Obj` of type `R`. There is only one reference direction defined, from the record denoted by `Obj` to the component protected object denoted by `Obj.Y`. This idiom, however, requires a reference in the opposite direction, from `Obj.Y` to `Obj`.

This may seem like an unrealistic situation, but it is not. An IO device type that involves interrupt handling is just one real-world example, one that we will show in detail.

The idiom context is a type because there will often be multiple real-world entities being represented in software. Representing these entities as multiple objects declared of a single type is by far the most reasonable approach.

We assume the functional unit will be implemented as an *Abstract Data Type (ADT)* (page 1187). Strictly speaking, the ADT idiom is not required here, but that is the best approach for defining major types, for the good reasons given in that idiom entry. There's no reason not to use an ADT in this case so we will.

38.2 Implementation(s)

As mentioned, the implementation approach applies to enclosed components of both task types and protected types. We will continue the discussion in terms of protected types.

The implementation has two parts:

1. An access discriminant on the PO type, designating the enclosing record's type. That part is straightforward.
2. A value given to that discriminant that designates the object of the enclosing record type, i.e., the record object that contains that PO. That part requires a relatively obscure language construct.

Given those two parts, the PO can then dereference its access discriminant to read or update the other components in the same enclosing record object.

Consider the following (very artificial) package declaration illustrating these two parts:

```
package P is
  type Device is tagged limited private;
private

  protected type Controller (Encloser : not null access Device) is
    -- Part 1

    procedure Increment_X;
  end Controller;

  type Device is tagged limited record
    X : Integer; -- arbitrary type

    C : Controller (Encloser => ...);
    -- Part 2, not fully shown yet
  end record;
end P;
```

The record type named Device contains a component named X, arbitrarily of type **Integer**, and another component C that is of protected type Controller. Part #1 of the implementation is the access discriminant on the declaration of the protected type Controller:

```
protected type Controller (Encloser : not null access Device) is
```

Given a value for the discriminant Encloser, the code within the spec and body of type Controller can then reference some Device object via that discriminant.

But not just any object of type Device will suffice. For Part #2, we must give the Encloser discriminant a value that refers to the current instance of the record object containing this same PO object. In the package declaration above, the value passed to Encloser is elided.

The following is that code again, now showing just the declaration for Device, but also including the construct that is actually passed. This is where the subtlety comes into play:

```
type Device is tagged limited record
    ...
    C : Controller (Encloser => Device'Access);
end record;
```

The subtlety is the expression Device'Access. Within a type declaration, usage of the type name denotes the current instance of that type. The current instance of a type is the object of the type that is associated with the execution that evaluates the type name. For example, during execution, when an object of type Device is elaborated, the name Device refers to that object.

It isn't compiler-defined magic, the semantics are defined by the Ada standard so it is completely portable. (There are other cases for expressing the current instance of task types, protected types, and generics.)

Therefore, within the declaration of type Device, the expression Device'Access provides an access value designating the current instance of that type. This is exactly what we want and is the crux of the idiom expression. With that discriminant value, the enclosed PO spec and body can reference the other record components of the same object that contains the PO.

To illustrate that, here is the package body for this trivial example. Note the value referenced in the body of procedure Increment_X:

```
package body P is
    protected body Controller is
        procedure Increment_X is
        begin
            Encloser.X := Encloser.X + 1;
        end Increment_X;
    end Controller;
end P;
```

Specifically, the body of procedure Increment_X can use the access discriminant Encloser to get to the current instance's X component. (We could express it as Encloser.all.X but why bother. Implicit dereferencing is a wonderful thing.)

That's the implementation. Now for some necessary details.

Note that we declared type Device as a limited type, first in the visible part of the package:

```
type Device is tagged limited private;
```

and again in the type completion in the package private part:

```
type Device is tagged limited record ... end record;
```

We declare Device as a limited type because we want to preclude assignment statements for client objects of the type. Assignment of the enclosing record object would leave the PO Encloser discriminant designating the prior (right-hand side) enclosing object. If the PO is written with the assumption that the enclosing object is always the one identified during creation of the PO, that assumption will no longer hold. We didn't state it up-front, but that is the assumption underlying the idiom as described, and in fact, only limited types may have a component that uses the Access attribute in this way. Also note that any type that includes a protected or task object is limited, so a type like Device will necessarily be limited in any case.

The type need not be tagged for this approach, but it must be limited in both its partial view and its full view. More generally, a tagged type must be limited in both views if it is limited in either view.

For the idiom implementation to be legal, the type's completion in the private part must not merely be limited, but actually *immutably limited*, meaning that it is always truly limited. There are various ways to make that happen (see [AARM22 7.5 \(8.1/3\)](#)³⁶¹) but the easiest way to is to include the reserved word **limited** in the type definition within the full view, as we did above. That is known as making the type *explicitly limited*. It turns out having a task or protected component also makes it immutably limited, so this requirement is naturally satisfied in this use case.

Why does the compiler require the type to be immutably limited?

Recall that a (non-tagged) private type need not be limited in both views. It can be limited in the partial client view but non-limited in its full view:

```
package Q is
  type T is limited private;
  -- the partial view for clients in package visible part
  ...
private
  type T is record -- the full view in the package private part
    ...
  end record;
end Q;
```

Clients must treat type Q.T as if it is limited, but Q.T isn't really limited because the full view defines reality. Clients simply have a more restricted view of the type than is really the case.

Types that are immutably limited are necessarily limited in all views. That's important because the current instance of the type given in type_name '[Access](#)' must be aliased for '[Access](#)' to be legal. But if the type's view could change between limited and not limited, its current instance would be aliased in some contexts and not aliased in others. To prevent that complexity, the language requires the type to be immutably limited so that the current instance of the type will be aliased in every view. In practice, we're working with record types and type extensions, so just make the full type definition explicitly limited and all will be well:

```
package Q is
  type T is limited private;
  ...
private
  type T is limited record
    ...
  end record;
end Q;
```

Then, as mentioned, you can choose whether the type will also be tagged.

38.3 Real-World Example

For a concrete, real-world example, suppose we have a serial IO device on an embedded target board. The device can be either a UART or [USART](#)³⁶². For the sake of brevity let's assume we have USARTs available.

Many boards have more than one USART resident, so it makes sense to represent them in software as instances of an ADT. This example uses the USART ADT supported in the

³⁶¹ <http://www.ada-auth.org/standards/22aarm/html/AA-7-5.html>

³⁶² https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

Ada Drivers Library (ADL)³⁶³ that is named, imaginatively, USART. (We don't show package STM32.USARTs, but you will see it referenced in the example's context clauses.) Each of these USART devices can support either a polling implementation or an interrupt-driven implementation. We will first define a basic USART ADT, and then extend that to a new one that works with interrupts.

At the most basic level, to work with a given USART device we must combine it with some other hardware, specifically the IO pins that connect it to the outside world. That combination will be represented by a new ADT, the type `Device` defined in package `Serial_IO`.

Any given `Serial_IO.Device` object will be associated permanently with one USART. Therefore, type `Device` will have a discriminant named `Transceiver` designating that USART object.

There are some low-level operations that a `Serial_IO.Device` will implement, such as initializing the hardware and setting the baud rate and so forth. We can also implement the hardware-oriented input and output routines in this package because both are independent of the polling or interrupt-driven designs.

Here's the resulting package declaration for the serial IO device ADT. Parts of the package are elided for simplicity (the full code is *at the end of this idiom entry* (page 1255)):

```
with STM32;          use STM32;
with STM32.GPIO;    use STM32.GPIO;
with STM32.USARTs;  use STM32.USARTs;
with HAL; -- the ADL's Hardware Abstraction Layer

package Serial_IO is

    type Device (Transceiver : not null access USART) is tagged limited private;

    procedure Initialize
        (This          : in out Device;
         Tx_Pin        : GPIO_Point;
         Rx_Pin        : GPIO_Point;
         ...);

    procedure Configure (This : in out Device; Baud_Rate : Baud_Rates; ...);
    ...
    procedure Put (This : in out Device; Data : HAL.UInt8) with Inline;
    procedure Get (This : in out Device; Data : out HAL.UInt8) with Inline;

private

    type Device (Transceiver : not null access USART) is tagged limited record
        Tx_Pin : GPIO_Point;
        Rx_Pin : GPIO_Point;
        ...
    end record;

end Serial_IO;
```

When called, procedure `Initialize` does the hardware setup required, such as enabling power for the USART and pins. We can ignore those details for this discussion.

Given this basic `Device` type we can then use inheritance (type extension) to define distinct types that support the polling and interrupt-driven facilities. These types will themselves be ADTs. Let's focus on the new interrupt-driven ADT, named `Serial_Port`. This type will be declared in the child package `Serial_IO.Interrupt_Driven`.

When interrupts are used, each USART raises a USART-specific interrupt for sending and receiving. Each interrupt occurrence is specific to one device. Therefore, the interrupt

³⁶³ https://github.com/AdaCore/Ada_Drivers_Library

handler code is specific to each `Serial_Port` object instance. We use protected objects as interrupt handlers in (canonical) Ada, hence each `Serial_Port` object will contain a dedicated interrupt handling PO as a record component.

As a controller and handler for a USART's interrupts, the PO will require a way to access the USART and pins being driven. Our idiom design provides that access.

Here is the client view of the ADT for the interrupt-driven implementation:

```
with Ada.Interrupts;      use Ada.Interrupts;
with HAL;
with System; use System;

package Serial_IO.Interrupt_Driven is

  type Serial_Port
    (Transceiver : not null access USART;
     IRQ         : Interrupt_ID;
     IRQ_Priority : Interrupt_Priority)
  is new Serial_IO.Device with private;

  -- The procedures Initialize and Configure, among others, are
  -- implicitly declared here as operations inherited from
  -- Serial_IO.Device.

  overriding
  procedure Put (This : in out Serial_Port; Data : HAL.UInt8)
    with Inline;

  overriding
  procedure Get (This : in out Serial_Port; Data : out HAL.UInt8)
    with Inline;

private
  ...
end Serial_IO.Interrupt_Driven;
```

The declaration of type `Serial_Port` uses *Interface Inheritance* (page 1237) to extend `Serial_IO.Device` with both visible and hidden components. The three visible extension components are the discriminants `Transceiver`, `IRQ`, and `IRQ_Priority`. `Transceiver` will designate the USART to drive (discussed in a moment). `IRQ` is the `Interrupt_ID` indicating the interrupt that the associated USART raises. `IRQ_Priority` is the priority for that interrupt. (*IRQ* is a common abbreviation for *Interrupt ReQuest*.) These two interrupt-oriented discriminants are used within the PO declaration to configure it for interrupt handling.

Clients will know which USART they are working with so they will be able to determine which interrupt ID and priority to specify, presumably by consulting the board documentation.

Now let's examine the `Serial_Port` type completion in the package's private part.

We've said we will use a PO interrupt handler as a component of the `Serial_Port` record type. This PO type, named `I0_Manager`, will include discriminants for the two interrupt-specific values it requires as an interrupt handler. It will also have a discriminant providing access to the enclosing `Serial_Port` record type.

```
protected type I0_Manager
  (IRQ         : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority;
   Port       : not null access Serial_Port)
with
  Interrupt_Priority => IRQ_Priority
is
  entry Put (Datum : HAL.UInt8);
```

(continues on next page)

(continued from previous page)

```

    entry Get (Datum : out HAL.UInt8);
private
    ...
    procedure IRQ_Handler with Attach_Handler => IRQ;
end IO_Manager;

```

Note how the first two discriminants are used within the type declaration to give the priority of the PO and to attach the interrupt handler procedure `IRQ_Handler` to the interrupt indicated by `IRQ`. The `Port` discriminant will be the back-reference to the enclosing record object.

We can then, finally, provide the `Serial_Port` type completion, in which the record object and protected object are connected whenever a `Serial_Port` object is declared:

```

type Serial_Port
  (Transceiver : not null access USART;
   IRQ         : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority)
is new Serial_IO.Device (Transceiver) with record
  Controller : IO_Manager (IRQ, IRQ_Priority, Serial_Port'Access);
end record;

```

The type completion repeats the declaration in the public part, up to the point where the `Serial_Port.Transceiver` discriminant is passed to the `Serial_IO.Device.Transceiver` discriminant. Type `Device` must be constrained with a discriminant value here, so we just pass the discriminant defined for `Serial_Port`.

Why does `Serial_Port` also have a `Transceiver` discriminant? Just as `Serial_IO.Device` is a complete wrapper for the combination of a USART and IO pins, `Serial_Port` is a stand-alone wrapper for `Serial_IO.Device`. Hence `Serial_Port` also needs a discriminant designating a USART to be complete.

The full definition of type `Serial_Port` contains the declaration of the component named `Controller`, of the protected type `IO_Manager`. The two interrupt-oriented discriminants from `Serial_Port` are passed to the discriminants defined for this PO component. The third `IO_Manager` discriminant value, `Serial_Port'Access`, denotes the current instance of the `Serial_Port` type. Thus the idiom requirements are achieved.

Let's see that back-reference in use within the protected body.

As mentioned, there is one interrupt used for both sending and receiving, per USART. Strictly speaking, the device itself does use two dedicated interrupts, one indicating that a 9-bit value has been received, and one indicating that transmission for a single 9-bit value has completed. But these two are signaled to the software on one interrupt line, and that is the value indicated by `IRQ`.

Therefore, there is one interrupt handling protected procedure, named `IRQ_Handler`. In response to this interrupt, `IRQ_Handler` determines which event has occurred by checking one of the `Transceiver` status registers. The back-reference through `Port` makes that possible. Other `Transceiver` routines are also called via `Port`, and `Port.all` is passed to the `Put` and `Get` calls:

```

procedure IRQ_Handler is
begin
  -- check for data arrival
  if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
    Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
  then -- handle reception
    -- call the Serial_IO.Device version:
    Get (Serial_IO.Device (Port.all), Incoming);

```

(continues on next page)

(continued from previous page)

```

    Await_Reception_Complete : loop
        exit when not Port.Transceiver.Status (Read_Data_Register_Not_Empty);
    end loop Await_Reception_Complete;
    Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
    Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
    Incoming_Data_Available := True;
end if;

-- check for transmission ready
if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
    Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
then -- handle transmission
    -- call the Serial_IO.Device version:
    Put (Serial_IO.Device (Port.all), Outgoing);

    Port.Transceiver.Disable_Interrupts (Transmission_Complete);
    Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
    Transmission_Pending := False;
end if;
end IRQ_Handler;

```

In this example, although the PO only accesses the Transceiver component in the enclosing record object, additional functionality might need to access more components, for this example perhaps using some of the inherited IO pin components.

38.4 Pros

The implementation is directly expressed, requiring only an access discriminant and the current instance semantics of type_name [Access](#).

Although the real-world example is complex — multiple discriminants are involved, and a type extension — the implementation itself requires little text. Interrupt handling is relatively complex in any language.

38.5 Cons

The record type must be truly a limited type, but that is not the severe limitation it was in earlier versions of Ada. Note that although access discriminants are required, there is no dynamic allocation involved.

38.6 Relationship With Other Idioms

This idiom is useful when we have a record type enclosing a PO or task object. If the [Abstract Data Machine \(ADM\)](#) (page 1193) would instead be appropriate, the necessary visibility can be achieved without requiring this implementation approach because there would be no enclosing record type. But as described in the ADM discussion, the [ADT approach](#) (page 1187) is usually superior.

38.7 Notes

As a wrapper abstraction for a USART, package `Serial_IO` is still more hardware-specific than absolutely necessary, as reflected in the parameters' types for procedure `Initialize` and the corresponding record component types. We could use the Hardware Abstraction

Layer (HAL) to further isolate the hardware dependencies, but that doesn't affect the idiom expression itself.

38.8 Full Source Code for Selected Units

We did not show some significant parts of the code discussed above, for the sake of not obscuring the points being made. Doing so, however, means that the interested reader cannot see how everything fits together and works, such as the actual IO using interrupts. The code below shows the relevant packages in their entirety. Note that the ADL STM32 hierarchy packages and the HAL (Hardware Abstraction Layer) package are in the [Ada Drivers Library on GitHub](https://github.com/AdaCore/Ada_Drivers_Library)³⁶⁴.

First, the basic Serial_IO abstraction:

```
with STM32;          use STM32;
with STM32.GPIO;     use STM32.GPIO;
with STM32.USARTs;   use STM32.USARTs;
with HAL;

package Serial_IO is

    type Device (Transceiver : not null access USART) is tagged limited private;

    procedure Initialize
        (This      : in out Device;
         Transceiver_AF : GPIO_Alternate_Function;
         Tx_Pin    : GPIO_Point;
         Rx_Pin    : GPIO_Point;
         CTS_Pin   : GPIO_Point;
         RTS_Pin   : GPIO_Point);
    -- must be called before Configure

    procedure Configure
        (This      : in out Device;
         Baud_Rate : Baud_Rates;
         Parity    : Parities    := No_Parity;
         Data_Bits : Word_Lengths := Word_Length_8;
         End_Bits  : Stop_Bits    := Stopbits_1;
         Control   : Flow_Control := No_Flow_Control);

    procedure Set_CTS (This : in out Device; Value : Boolean) with Inline;
    procedure Set_RTS (This : in out Device; Value : Boolean) with Inline;

    procedure Put (This : in out Device; Data : HAL.UInt8) with Inline;
    procedure Get (This : in out Device; Data : out HAL.UInt8) with Inline;

private

    type Device (Transceiver : not null access USART) is tagged limited record
        Tx_Pin : GPIO_Point;
        Rx_Pin : GPIO_Point;
        CTS_Pin : GPIO_Point;
        RTS_Pin : GPIO_Point;
    end record;

end Serial_IO;
```

And the package body:

³⁶⁴ https://github.com/AdaCore/Ada_Drivers_Library

```

with STM32.Device; use STM32.Device;

package body Serial_IO is

    -----
    -- Initialize --
    -----

    procedure Initialize
    (This      : in out Device;
     Transceiver_AF : GPIO_Alternate_Function;
     Tx_Pin    : GPIO_Point;
     Rx_Pin    : GPIO_Point;
     CTS_Pin   : GPIO_Point;
     RTS_Pin   : GPIO_Point)
    is
        IO_Pins : constant GPIO_Points := Rx_Pin & Tx_Pin;
    begin
        This.Tx_Pin := Tx_Pin;
        This.Rx_Pin := Rx_Pin;
        This.CTS_Pin := CTS_Pin;
        This.RTS_Pin := RTS_Pin;

        Enable_Clock (This.Transceiver.all);

        Enable_Clock (IO_Pins);

        Configure_IO
        (IO_Pins,
         Config => (Mode_AF,
                    AF => Transceiver_AF,
                    AF_Speed => Speed_50MHz,
                    AF_Output_Type => Push_Pull,
                    Resistors => Pull_Up));

        Enable_Clock (RTS_Pin & CTS_Pin);

        Configure_IO (RTS_Pin, Config => (Mode_In, Resistors => Pull_Up));

        Configure_IO
        (CTS_Pin,
         Config => (Mode_Out,
                    Speed => Speed_50MHz,
                    Output_Type => Push_Pull,
                    Resistors => Pull_Up));
    end Initialize;

    -----
    -- Configure --
    -----

    procedure Configure
    (This      : in out Device;
     Baud_Rate : Baud_Rates;
     Parity    : Parities    := No_Parity;
     Data_Bits : Word_Lengths := Word_Length_8;
     End_Bits  : Stop_Bits    := Stopbits_1;
     Control   : Flow_Control := No_Flow_Control)
    is
    begin
        This.Transceiver.Disable;
    end Configure;

```

(continues on next page)

(continued from previous page)

```

    This.Transceiver.Set_Baud_Rate    (Baud_Rate);
    This.Transceiver.Set_Mode         (Tx_Rx_Mode);
    This.Transceiver.Set_Stop_Bits    (End_Bits);
    This.Transceiver.Set_Word_Length  (Data_Bits);
    This.Transceiver.Set_Parity       (Parity);
    This.Transceiver.Set_Flow_Control (Control);

    This.Transceiver.Enable;
end Configure;

-----
-- Set_CTS --
-----

procedure Set_CTS (This : in out Device; Value : Boolean) is
begin
    This.CTS_Pin.Drive (Value);
end Set_CTS;

-----
-- Set_RTS --
-----

procedure Set_RTS (This : in out Device; Value : Boolean) is
begin
    This.RTS_Pin.Drive (Value);
end Set_RTS;

-----
-- Put --
-----

procedure Put (This : in out Device; Data : HAL.UInt8) is
begin
    This.Transceiver.Transmit (HAL.UInt9 (Data));
end Put;

-----
-- Get --
-----

procedure Get (This : in out Device; Data : out HAL.UInt8) is
    Received : HAL.UInt9;
begin
    This.Transceiver.Receive (Received);
    Data := HAL.UInt8 (Received);
end Get;

end Serial_IO;

```

Next, the interrupt-driven extension.

```

with Ada.Interrupts;      use Ada.Interrupts;
with HAL;
with System; use System;

package Serial_IO.Interrupt_Driven is
    pragma Elaborate_Body;

    type Serial_Port
        (Transceiver : not null access USART;

```

(continues on next page)

(continued from previous page)

```

    IRQ          : Interrupt_ID;
    IRQ_Priority : Interrupt_Priority)
is new Serial_IO.Device with private;
-- A serial port that uses interrupts for I/O. Extends the Device
-- abstraction that is itself a wrapper for the USARTs hardware.

-- The procedures Initialize and Configure, among others, are implicitly
-- declared here, as operations inherited from Serial_IO.Device

overriding
procedure Put (This : in out Serial_Port; Data : HAL.UInt8)
with Inline;
-- Non-blocking, ie the caller can return before the Data goes out,
-- but does block until the underlying UART is not doing any other
-- transmitting. Does no polling. Will not interfere with any other I/O
-- on the same device.

overriding
procedure Get (This : in out Serial_Port; Data : out HAL.UInt8)
with Inline;
-- Blocks the caller until a character is available! Does no polling.
-- Will not interfere with any other I/O on the same device.

```

private

```

-- The protected type defining the interrupt-based I/O for sending and
-- receiving via the USART attached to the serial port designated by
-- Port. Each serial port object of the type defined by this package has
-- a component of this protected type.

```

```

protected type IO_Manager
  (IRQ          : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority;
   Port         : not null access Serial_Port)
-- with
--   Interrupt_Priority => IRQ_Priority -- compiler bug :-(
is
  pragma Interrupt_Priority (IRQ_Priority);

  entry Put (Datum : HAL.UInt8);

  entry Get (Datum : out HAL.UInt8);

```

private

```

    Outgoing : HAL.UInt8;
    Incoming  : HAL.UInt8;

    Incoming_Data_Available : Boolean := False;
    Transmission_Pending    : Boolean := False;

    procedure IRQ_Handler with Attach_Handler => IRQ;

end IO_Manager;

type Serial_Port
  (Transceiver : not null access USART;
   IRQ          : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority)
is
  new Serial_IO.Device (Transceiver) with
  record

```

(continues on next page)

(continued from previous page)

```

    Controller : IO_Manager (IRQ, IRQ_Priority, Serial_Port'Access);
    -- Note that Serial_Port'Access provides the Controller with a view
    -- to the current instance's components, including the discriminant
    -- components
end record;

end Serial_IO.Interrupt_Driven;

```

And the package body:

```

with STM32.Device; use STM32.Device;

package body Serial_IO.Interrupt_Driven is

    -----
    -- Put --
    -----

    overriding
    procedure Put (This : in out Serial_Port; Data : HAL.UInt8) is
    begin
        This.Controller.Put (Data);
    end Put;

    -----
    -- Get --
    -----

    overriding
    procedure Get (This : in out Serial_Port; Data : out HAL.UInt8) is
    begin
        This.Transceiver.Enable_Interrupts (Received_Data_Not_Empty);
        This.Controller.Get (Data);
    end Get;

    -----
    -- IO_Manager --
    -----

    protected body IO_Manager is

        -----
        -- IRQ_Handler --
        -----

        procedure IRQ_Handler is
        begin
            -- check for data arrival
            if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
                Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
            then -- handle reception
                -- call the Serial_IO.Device version:
                Get (Serial_IO.Device (Port.all), Incoming);

                Await_Reception_Complete : loop
                    exit when not
                        Port.Transceiver.Status (Read_Data_Register_Not_Empty);
                end loop Await_Reception_Complete;
                Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
                Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
                Incoming_Data_Available := True;
            end if;
        end IRQ_Handler;
    end protected body IO_Manager;

```

(continues on next page)

(continued from previous page)

```

end if;

-- check for transmission ready
if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
   Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
then -- handle transmission
   -- call the Serial_IO.Device version:
   Put (Serial_IO.Device (Port.all), Outgoing);

   Port.Transceiver.Disable_Interrupts (Transmission_Complete);
   Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
   Transmission_Pending := False;
end if;
end IRQ_Handler;
-----
-- Put --
-----

entry Put (Datum : HAL.UInt8) when not Transmission_Pending is
begin
   Transmission_Pending := True;
   Outgoing := Datum;
   Port.Transceiver.Enable_Interrupts (Transmission_Complete);
end Put;

-----
-- Get --
-----

entry Get (Datum : out HAL.UInt8) when Incoming_Data_Available is
begin
   Datum := Incoming;
   Incoming_Data_Available := False;
end Get;

end IO_Manager;

end Serial_IO.Interrupt_Driven;

```

INTERRUPT HANDLING

39.1 Motivation

Recall that, in Ada, protected procedures are the standard interrupt-handling mechanism. The canonical interrupt handling and management model is defined in the [Systems Programming Annex, section C.3 of the Reference Manual](#)³⁶⁵. We assume that this optional annex is supported, and indeed effectively all compilers do support it. Likewise, we assume that the [Real-Time Annex, annex D](#)³⁶⁶, is supported (which would require [Annex C](#)³⁶⁷ to be supported anyway). Finally, we assume that either the Ravenscar or the Jorvik usage profile is applied. These two profiles define configurations of the two annexes that are appropriate for typical embedded systems that handle interrupts.

The definition of a canonical model mitigates differences imposed by the target, but some remain. For example, the number of different priority values, including interrupt priorities, differs with the targets involved. The model supports blocking of those interrupts at a lower priority than the currently executing interrupt handler, but the hardware might not support that behavior, although many do. None of these variations affect the expression of the idioms themselves.

The response to interrupts is often arranged in logical levels. The first level is the protected procedure handler itself. In some cases, everything required to handle the interrupt is performed there. However, some applications require more extensive, asynchronous processing of the data produced by the first level interrupt handler. In this case a second-level response can be defined, consisting of a task triggered by the first level. For example, the interrupt handler could respond to the first arrival of a character on a [USART](#)³⁶⁸, poll for the remainder (or not), and then notify a task to perform analysis of the entire string received.

But even if no second-level interrupt processing is required, the interrupt handler may be required to notify the application that the event has occurred. Because interrupts are asynchronous, and logically concurrent with the application code, the association of an application task to a given interrupt-driven event is convenient and common.

Hence a task is often involved. How the handler procedure notifies the task leads to a couple of different idiom implementations. In both cases notification amounts to releasing the previously suspended task for further execution.

In the following section we show how to express these three idioms: one for using protected procedures alone, and two in which a protected procedure handler notifies a task.

³⁶⁵ <http://www.ada-auth.org/standards/12rm/html/RM-C-3.html>

³⁶⁶ <http://www.ada-auth.org/standards/12rm/html/RM-D.html>

³⁶⁷ <http://www.ada-auth.org/standards/12rm/html/RM-C.html>

³⁶⁸ https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

39.2 Implementation(s)

39.2.1 First Level Handler Alone

In this approach the interrupt handler protected procedure does everything necessary and does not require a second-level handler.

An interrupt handler that simply copies data from one location to another is a good example of a necessary and sufficient first-level handler. The enclosing application assumes the copying is occurring as required and needs no explicit notification. If the copying isn't happening the failure will be obvious.

So, given that, why discuss such a scenario? Two reasons: to show how it is done in general, and especially, to show how double-buffering can be implemented very elegantly with interrupts.

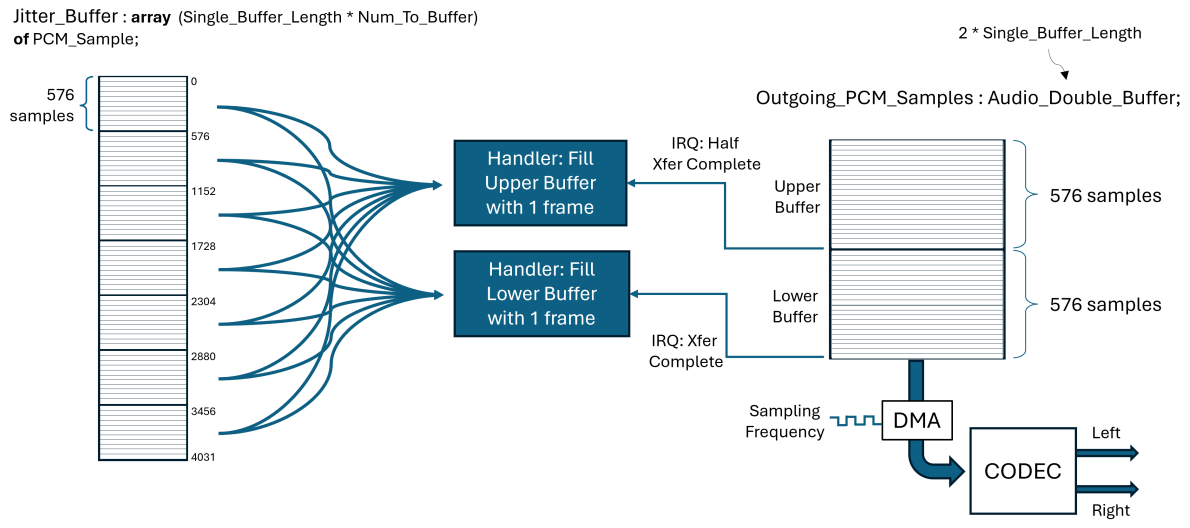
For a concrete example, consider an audio streaming device that takes PCM samples from Ethernet via incoming UDP packets and transfers them to an [audio CODEC device](#)³⁶⁹ on the target board. The CODEC output is physically connected to a high-quality amplifier and speakers. No upper-level application thread requires notification of the copying: if the transfer is working the audio output occurs, otherwise it does not.

In our implementation the CODEC device is *fed* from a buffer named `Outgoing_PCM_Samples`. The buffer must always have new samples available when the CODEC is ready for them, because delays or breaks would introduce audible artifacts. The timing is determined by the sampling rate used by the audio source, prior to transmission. To match that rate and to provide it efficiently, we use DMA to transfer the data from the buffer to the CODEC. In addition, `Outgoing_PCM_Samples` is double-buffered to help ensure the samples are always available upon demand.

However, the incoming UDP packets don't arrive at exact intervals. Because of this *jitter* in the arrival times, we cannot directly insert the PCM samples from these incoming packets into the `Outgoing_PCM_Samples` buffer. The delays would be audible. Therefore, we use a *jitter buffer* to deal with the arrival time variations. This jitter buffer holds the PCM samples as they arrive in the UDP packets, in sufficient amounts to de-couple the arrival time jitter from the outgoing data. A jitter buffer can do much more than this, such as correcting the order of arriving packets, but in this specific case the additional functionality is not required.

We use two DMA interrupts to copy data from the jitter buffer to the `Outgoing_PCM_Samples` buffer. The rationale for using two interrupts, rather than one, is given momentarily. The figure below illustrates the overall approach, with the jitter buffer on the left, the two interrupt handlers in the middle, and the `Outgoing_PCM_Samples` buffer on the right.

³⁶⁹ https://en.wikipedia.org/wiki/Audio_codec



Each UDP packet contains 576 PCM samples, used as the *single buffer length* for the double-buffered `Outgoing_PCM_Samples` and the `Jitter_Buffer`.

The advantage of double-buffering is that the producer can be filling one buffer while the consumer is removing data from the other. These directions switch when the current output buffer becomes empty. The result is a fast, continuous output stream. Many audio and video devices use double-buffering for that reason.

To express double-buffering you could use two physically distinct array objects, switching between them when the DMA controller signals that the current outgoing buffer is empty. That would require keeping track of which buffer is being filled and which is being emptied. There is an elegant, simpler alternative that uses two different DMA interrupts instead of one. (The DMA device must support this approach directly.)

In this alternative, there is one physical array (`Outgoing_PCM_Samples`), containing twice the number of components as a single physical buffer would contain. We can then use the two interrupts to treat the one physical array as two logical buffers.

The two DMA interrupts are triggered as the DMA transfer consumes the content within this single array. One interrupt is triggered when the transfer reaches the physical half-way point in the array. The other interrupt is triggered when the transfer reaches the physical end of the array. Therefore, because the array is twice the size of a single buffer, each interrupt corresponds to one of the two logical buffers becoming empty.

Furthermore, the DMA device generating these interrupts is configured so that it does not stop. After triggering the *half transfer complete* interrupt the DMA continues reading, now from the second logical buffer. After triggering the *transfer complete* interrupt the DMA device starts over at the beginning of the array, reading from the first logical buffer again.

Therefore, we have two distinct interrupt handlers, one for each of the two interrupts. When the *half transfer complete* handler is invoked, the upper logical buffer is now empty, so the handler for that half fills it. Likewise, the *transfer complete* interrupt handler fills the lower logical buffer at the bottom half of the array. There's no need to keep track of which buffer is being filled or emptied. It's all being emptied, and the handlers always fill the same upper or lower halves of the array. As long as each handler completes filling their half before the DMA transfer begins reading it, all is well.

Here's the declaration of the protected object containing the DMA interrupt handling code.

```
protected DMA_Interrupt_Controller with
  Interrupt_Priority => DMA_Interrupt_Priority
is
private
```

(continues on next page)

(continued from previous page)

```

procedure DMA_IRQ_Handler with
  Attach_Handler => STM32.Board.Audio_Out_DMA_Interrupt;
end DMA_Interrupt_Controller;

```

A few points are worth highlighting.

First, `DMA_Interrupt_Priority` is an application-defined constant. The actual value isn't important to this discussion. The handler procedure is attached to an interrupt that is specific to the target board, so it is defined in the package `STM32.Board` in the Ada Drivers Library. Each target board supported by the library has such a package, always with the same package name. This particular STM32 board has dedicated audio DMA support, along with the CODEC.

Second, there's nothing declared in the visible part of the PO. More to the point, everything is declared in the optional private part. That placement is a matter of style, but it's good style. No software client should ever call the protected procedure — only the hardware should call it, via the runtime library — so we make it impossible for any client to call it accidentally. That placement also informs the reader of our intent.

Third, we said there are two interrupts, but only one interrupt handler procedure is declared and attached. There's nothing inherently wrong with one routine handling multiple interrupts, although conceptually it is not ideal. In this case it is necessary because on this target both device interrupts arrive at the MCU on one external interrupt line. Therefore, the one protected procedure handler handles both device interrupts, querying the DMA status flags to see which interrupt is active. This approach is shown below. Note that there must be an enclosing package, with multiple context clauses, but we do not show them so that we can focus on the interrupt handler itself.

```

protected body DMA_Interrupt_Controller is

  procedure DMA_IRQ_Handler is
    use STM32.Board; -- for the audio DMA
  begin
    if Status (Audio_DMA,
               Audio_DMA_Out_Stream,
               DMA.Half_Transfer_Complete_Indicated)
    then
      -- The middle of the double-buffer array has been reached by the
      -- DMA transfer, therefore the "upper half buffer" is empty.
      Fill_Logical_Buffer (Outgoing_PCM_Samples,
                           Starting_Index => Upper_Buffer_Start);
      Clear_Status (Audio_DMA,
                    Audio_DMA_Out_Stream,
                    DMA.Half_Transfer_Complete_Indicated);
    end if;

    if Status (Audio_DMA,
               Audio_DMA_Out_Stream,
               DMA.Transfer_Complete_Indicated)
    then
      -- The bottom of the double-buffer array has been reached by the
      -- DMA transfer, therefore the "lower half buffer" is empty.
      Fill_Logical_Buffer (Outgoing_PCM_Samples,
                           Starting_Index => Lower_Buffer_Start);
      Clear_Status (Audio_DMA,
                    Audio_DMA_Out_Stream,
                    DMA.Transfer_Complete_Indicated);
    end if;
  end DMA_IRQ_Handler;
end DMA_Interrupt_Controller;

```

In both cases `Fill_Logical_Buffer` is called to insert samples from the jitter buffer into one of the logical buffers. The difference is the value passed to the formal parameter `Starting_Index`. This is the array index at which filling should begin within `Outgoing_PCM_Samples`. `Upper_Buffer_Start` corresponds to `Outgoing_PCM_Samples'First`, and `Lower_Buffer_Start` is `Outgoing_PCM_Samples'First + Single_Buffer_Length`.

That's all the software has to do. Offloading work to the hardware, in this case the DMA controller, is always a good idea, but that's especially true for less powerful targets, e.g., microcontrollers. Note that the availability of the *half transfer complete* interrupt varies across different DMA devices.

The implementation of `Fill_Logical_Buffer` is straightforward and need not be shown. However, the procedure declares a local variable named `Incoming_PCM_Samples` that has ramifications worth noting. In particular, the representation may require altering and rebuilding the underlying Ada run-time library.

The object `Incoming_PCM_Samples` is declared within `Fill_Logical_Buffer` like so:

```
Incoming_PCM_Samples : Jitter_Buffer.Sample_Buffer_Slice;
```

The alteration might be required because `Fill_Logical_Buffer` executes entirely in the interrupt handler procedure's context. Hence the storage used by the procedure's execution comes from the interrupt handler stack. Interrupt handlers typically do relatively little, and, as a result, a relatively small stack allocation is typically defined for them. The storage for `Incoming_PCM_Samples` might exceed that allocation.

Specifically, we said that `Fill_Logical_Buffer` fills an entire half of the double-buffer, i.e., it works in terms of `Single_Buffer_Length`. If `Sample_Buffer_Slice` is an actual array, the required storage might be considerable.

The interrupt stack allocation is set by the run-time library source code in GNAT, as is common. You could increase the allocation and rebuild the run-time.

On the other hand, `Sample_Buffer_Slice` need not be an actual array. It could be a record type containing a (read-only) pointer to the jitter buffer array and an index indicating where in that array the *slice* to be transferred begins. That representation would obviously require much less stack space, obviating the run-time library change and rebuild. Moreover, that representation would allow `Fill_Logical_Buffer` to copy directly from the jitter buffer into the final destination, i.e., `Outgoing_PCM_Samples`. If `Incoming_PCM_Samples` is an array, we'd have to copy from the jitter buffer into `Incoming_PCM_Samples`, and then again from there to `Outgoing_PCM_Samples`. That's an extra copy operation we can avoid.

A related issue, perhaps requiring a run-time change, is the *secondary stack* allocation for interrupt handlers. The secondary stack is a common approach to implementing calls to functions that return values of unconstrained subtypes (usually, unconstrained array types, such as `String`). Because the result size is not known at the point of the call, using the primary call stack for holding the returned value is messy. The function's returned value would follow the stack space used for the call itself. But on return, only the call space is popped, leaving a *hole* in the stack because the value returned from the function remains on the stack. Therefore, another separate stack is commonly used for these functions. (GNAT does so.) The interrupt handler code could exhaust this allocation as well. The allocation amount is also specified in the run-time library source code. But, as with the situation above, the source code can be changed, in this case to avoid calling functions with unconstrained result types. The trade-off is whether that change is more costly than changing and rebuilding the run-time, as well as maintaining the change.

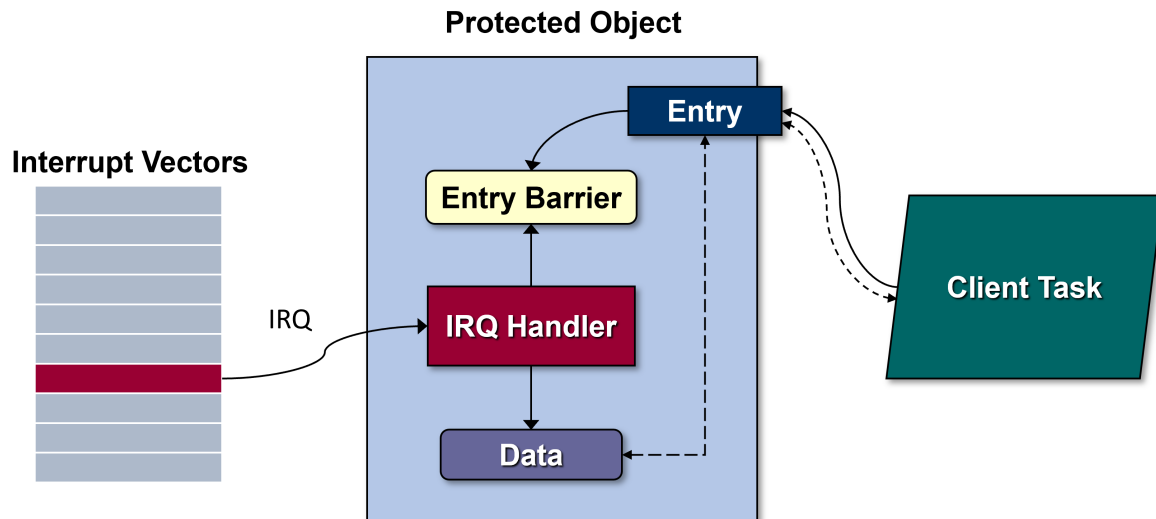
39.2.2 Task Notification Introduction

The first idiom implementation did not require notifying a task, but these next idiom implementations will do so. As we mentioned earlier, how the interrupt handler achieves this notification leads to two distinct idioms. Ultimately the difference between them is whether

or not the interrupt handler must communicate with the task. In both cases the handler synchronizes with the task because of the notification required.

39.2.3 Task Notification With Communication

In this implementation the interrupt handler releases a task but also communicates with it when doing so. Therefore, a protected object is used, and the entry parameters are the communication medium. The approach is depicted in the figure below:



The interrupt handler stores data within the PO and only enables the entry barrier when ready to either produce it or consume it via the entry parameters. The dashed lines in the figure represent this data flow.

By coincidence, this is the notification approach used in the idiom entry *Providing Component Access to Enclosing Record Objects* (page 1247). In that implementation, client tasks call two entries to Put and Get single characters, so the data stored in the PO consists of those characters. We did not mention it there because we were focused on that other idiom, i.e., how to give visibility within a PO/task component to an enclosing record object.

Be sure to understand the code for the other idiom before exploring this one. We will repeat elided parts of the code and only discuss the parts relevant for this current idiom. Because we are focused now on the interrupt handling task notification, here is the full interrupt handler PO type declaration — `IO_Manager` — within the elided package declaration:

```
package Serial_IO.Interrupt_Driven is

  type Serial_Port ... is new Serial_IO.Device with private;

  overriding
  procedure Put (This : in out Serial_Port; Data : HAL.UInt8)
    with Inline;

  overriding
  procedure Get (This : in out Serial_Port; Data : out HAL.UInt8)
    with Inline;

private

  protected type IO_Manager
    (IRQ           : Interrupt_ID;
     IRQ_Priority : Interrupt_Priority;
     Port         : not null access Serial_Port)
```

(continues on next page)

(continued from previous page)

```

with
  Interrupt_Priority => IRQ_Priority
is
  entry Put (Datum : HAL.UInt8);

  entry Get (Datum : out HAL.UInt8);

private

  Outgoing : HAL.UInt8;
  Incoming : HAL.UInt8;

  Incoming_Data_Available : Boolean := False;
  Transmission_Pending    : Boolean := False;

  procedure IRQ_Handler with Attach_Handler => IRQ;

end I0_Manager;

...

end Serial_I0.Interrupt_Driven;

```

A protected object of type `I0_Manager` is given a discriminant value that designates the enclosing `Serial_Port` object because that `Serial_Port` has the USART device required to do the actual I/O. The other two discriminants are required for configuring the interrupt handler and attaching it to the interrupt hardware.

The two octets `Outgoing` and `Incoming` are the values sent and received via the interrupt handler's manipulation of the USART. (A USART doesn't receive characters, as such, and we're ignoring the fact that it may work with a 9-bit value instead.)

The two Boolean components `Incoming_Data_Available` and `Transmission_Pending` are used for the two barrier expressions. Their purpose is explained below.

The bodies of visible procedures `Put` and `Get` (shown below) call through to the interrupt manager's protected entries, also named `Put` and `Get`. Those entries block the callers until the interrupt manager is ready for them, via the entry barriers controlled by the interrupt handler.

```

with STM32.Device; use STM32.Device;

package body Serial_I0.Interrupt_Driven is

  -----
  -- Put --
  -----

  overriding
  procedure Put (This : in out Serial_Port; Data : HAL.UInt8) is
  begin
    This.Controller.Put (Data);
  end Put;

  -----
  -- Get --
  -----

  overriding
  procedure Get (This : in out Serial_Port; Data : out HAL.UInt8) is

```

(continues on next page)

(continued from previous page)

```

begin
  This.Transceiver.Enable_Interrupts (Received_Data_Not_Empty);
  This.Controller.Get (Data);
end Get;

-----
-- IO_Manager --
-----

protected body IO_Manager is

  -----
  -- IRQ_Handler --
  -----

  procedure IRQ_Handler is
  begin
    -- check for data arrival
    if Port.Transceiver.Status (Read_Data_Register_Not_Empty) and then
      Port.Transceiver.Interrupt_Enabled (Received_Data_Not_Empty)
    then -- handle reception
      Get (Serial_IO.Device (Port.all), Incoming);
      -- call the Serial_IO.Device version!
      Await_Reception_Complete : loop
        exit when not
          Port.Transceiver.Status (Read_Data_Register_Not_Empty);
      end loop Await_Reception_Complete;
      Port.Transceiver.Disable_Interrupts (Received_Data_Not_Empty);
      Port.Transceiver.Clear_Status (Read_Data_Register_Not_Empty);
      Incoming_Data_Available := True;
    end if;

    -- check for transmission ready
    if Port.Transceiver.Status (Transmission_Complete_Indicated) and then
      Port.Transceiver.Interrupt_Enabled (Transmission_Complete)
    then -- handle transmission
      Put (Serial_IO.Device (Port.all), Outgoing);
      -- call the Serial_IO.Device version!
      Port.Transceiver.Disable_Interrupts (Transmission_Complete);
      Port.Transceiver.Clear_Status (Transmission_Complete_Indicated);
      Transmission_Pending := False;
    end if;
  end IRQ_Handler;

  -----
  -- Put --
  -----

  entry Put (Datum : HAL.UInt8) when not Transmission_Pending is
  begin
    Transmission_Pending := True;
    Outgoing := Datum;
    Port.Transceiver.Enable_Interrupts (Transmission_Complete);
  end Put;

  -----
  -- Get --
  -----

  entry Get (Datum : out HAL.UInt8) when Incoming_Data_Available is
  begin

```

(continues on next page)

(continued from previous page)

```

    Datum := Incoming;
    Incoming_Data_Available := False;
end Get;

end IO_Manager;

end Serial_IO.Interrupt_Driven;

```

Note how `IRQ_Handler` checks for which interrupt is active, possibly both, does whatever is necessary for that to be handled, and then sets the entry barriers accordingly. The barrier expression `Transmission_Pending` blocks `Put` callers until the current transmission, if any, completes. The barrier `Incoming_Data_Available` blocks `Get` callers until a character has been received and can be provided to the caller. The entry bodies copy the entry formal parameters to/from the internally stored characters and likewise set the entry barriers.

Note too how the body of procedure `Get` first enables the *received data available* interrupt before calling the entry. The body of the entry `Put` does something similar. They both work in concert with the handler procedure to manage the interrupts as required.

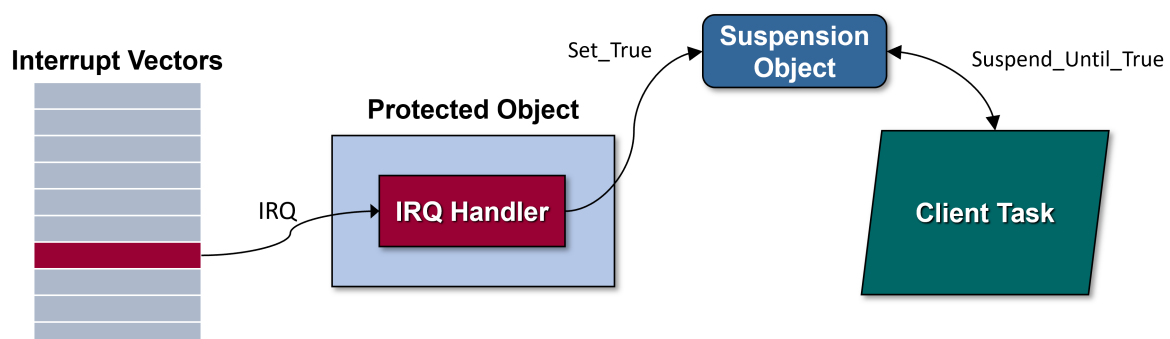
Using protected entries is ideal for this case because, after all, that is exactly what they are designed to do. Note that declaring multiple protected entries in a single protected type/object requires the Jorvik usage profile to be applied.

39.2.4 Task Notification Without Communication

In this implementation, the interrupt handler procedure is not required to communicate with the task. It only needs to synchronize with it, to release it.

Therefore, we can use a `Suspension_Object`: a language-defined, thread-safe *binary flag* type defined in package `Ada.Synchronous_Task_Control`. Objects of this type have two values: `True` and `False`, with `False` as the default initial value. There are two primary primitive operations: procedures `Suspend_Until_True` and `Set_True`. Procedure `Set_True` does just what you think it does. Procedure `Suspend_Until_True` suspends the caller (task) until the value of the specified argument becomes `True`, at which point the suspended task is allowed to continue execution. (Of course, if it was already `True` when `Suspend_Until_True` was called, the caller returns without suspending.) Critically, procedure `Suspend_Until_True` also sets the argument back to `False` before returning. As a result, those are the only two routines you're likely to need, although there are others.

The interrupt handler procedure in this approach simply calls `Set_True` for a `Suspension_Object` (an object of that type) visible to both the handler and the task. This arrangement is illustrated by the following figure:



The language requires the run-time library implementation to allow calls to `Set_False` and `Set_True` during any protected action, even one that has its ceiling priority in the `Interrupt_Priority` range, so this approach will work for interrupt handlers as well as tasks.

For our example we implement a facility for sending and receiving *messages* over a serial port, using interrupts. The design is similar to the implementation we just explored, and thus to the *Providing Component Access to Enclosing Record Objects* (page 1247) idiom. In that implementation, however, only single characters were sent and received, whereas messages will consist of one or more characters. Although there are differences, we assume that you are familiar enough with that idiom's approach that we don't need to go into all the details of the serial I/O, the USART, or the interrupt handler within a PO. We'll focus instead of the differences due to this idiom.

In this version we want to notify a task when an entire message has been sent or received, not just a single character. We'll define a message as a **String** that has a message-specified logical terminator character, e.g., the *nul* character. Transmission will cease when the terminator character is encountered when sending a message object. Similarly, a message is considered complete when the terminator character is received. (The terminator is not stored in message content.)

In a sense the interrupt handler is again communicating with tasks, but not directly, so entry parameters aren't applicable. Therefore, a `Suspension_Object` component is appropriate. But instead of one `Suspension_Object` variable, each `Message` object will contain two: one for notification of new content receipt, and one for notification of successful content transmission.

For the sake of the Separation of Concerns principle, the type `Message` should be an ADT of its own, in a dedicated package:

```
with Serial_IO;                               use Serial_IO;
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;

package Message_Buffers is

  type Message (Physical_Size : Positive) is tagged limited private;

  function Content (This : Message) return String;
  function Length (This : Message) return Natural;
  procedure Set (This : in out Message; To : String) with
    Pre => To'Length <= This.Physical_Size,
    Post => Length (This) = To'Length and Content (This) = To;
  ...
  function Terminator (This : Message) return Character;
  procedure Await_Transmission_Complete (This : in out Message);
  procedure Await_Reception_Complete (This : in out Message);
  procedure Signal_Transmission_Complete (This : in out Message);
  procedure Signal_Reception_Complete (This : in out Message);

private

  type Message (Physical_Size : Positive) is tagged limited record
    Content      : String (1 .. Physical_Size);
    Length       : Natural := 0;
    Reception_Complete : Suspension_Object;
    Transmission_Complete : Suspension_Object;
    Terminator    : Character := ASCII.NUL;
  end record;

end Message_Buffers;
```

In essence, a `Message` object is just the usual *variable length string* abstraction with a known terminator and ways to suspend and resume clients using them. Note the two `Suspension_Object` components.

In this example the tasks to be notified are application tasks rather than second-level interrupt handlers. Client tasks can suspend themselves to await either transmission completion or reception completion. The `Message` procedures simply call the appropriate routines for

the parameter's `Suspension_Object` components:

```
procedure Await_Transmission_Complete (This : in out Message) is
begin
  Suspend_Until_True (This.Transmission_Complete);
end Await_Transmission_Complete;
```

and likewise:

```
procedure Await_Reception_Complete (This : in out Message) is
begin
  Suspend_Until_True (This.Reception_Complete);
end Await_Reception_Complete;
```

The client task could look like the following, in this case the main program's environment task:

```
procedure Demo_Serial_Port_Nonblocking is

  Incoming : aliased Message (Physical_Size => 1024); -- arbitrary size
  Outgoing : aliased Message (Physical_Size => 1024); -- arbitrary size

  procedure Send (This : String) is
  begin
    Set (Outgoing, To => This);
    Start_Sending (COM, Outgoing'Unchecked_Access);
    Outgoing.Await_Transmission_Complete;
  end Send;

begin
  Initialize_Hardware (COM);
  Configure (COM, Baud_Rate => 115_200);

  Incoming.Set_Terminator (ASCII.CR);
  Send ("Enter text, terminated by CR.");
  loop
    Start_Receiving (COM, Incoming'Unchecked_Access);
    Incoming.Await_Reception_Complete;
    Send ("Received : " & Incoming.Content);
  end loop;
end Demo_Serial_Port_Nonblocking;
```

We don't show all the context clauses, for brevity, but one of the packages declares `COM` as the serial port. This demo doesn't exploit the nonblocking aspect because it does not perform any other actions before suspending itself after initiating sending and receiving. But it could do so, while the I/O is happening, only later suspending to await completion of the requested operation.

The interrupt handler procedure can signal both transmission and reception completion using the two other procedures:

```
procedure Signal_Transmission_Complete (This : in out Message) is
begin
  Set_True (This.Transmission_Complete);
end Signal_Transmission_Complete;

procedure Signal_Reception_Complete (This : in out Message) is
begin
  Set_True (This.Reception_Complete);
end Signal_Reception_Complete;
```

In this version of the Serial IO facility, the interrupt handler's enclosing protected type is the type `Serial_Port` itself, rather than a PO enclosed by a record type:

```
protected type Serial_Port
  (Device      : not null access Peripheral_Descriptor;
   IRQ         : Interrupt_ID;
   IRQ_Priority : Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is

  procedure Start_Sending (Msg : not null access Message);
  procedure Start_Receiving (Msg : not null access Message);

private

  Next_Out      : Positive;
  Outgoing_Msg  : access Message;
  Incoming_Msg  : access Message;

  procedure Handle_Transmission with Inline;
  procedure Handle_Reception    with Inline;
  procedure ISR with Attach_Handler => IRQ;

end Serial_Port;
```

Procedure *ISR (Interrupt Service Routine)* is the handler.

The two visible protected procedures, *Start_Sending* and *Start_Receiving*, are given non-null arguments when called (indirectly) by client tasks. Each argument is an access value designating a *Message* object declared by clients. The pointers are copied into the internal components, i.e., *Outgoing_Msg* and *Incoming_Msg*, for use by the interrupt handler procedure.

As with the earlier idiom above, there are multiple device interrupts, but they are all delivered on one external interrupt line. The handler procedure checks the status flags to see which interrupts are active and calls dedicated internal procedures accordingly. We don't need to see this infrastructure code again, so we can focus instead on one, the internal *Handle_Reception* procedure. The routine for transmitting is similar.

```
procedure Handle_Reception is
  Received_Char : constant Character :=
    Character'Val (Current_Input (Device.Transceiver.all));
begin
  if Received_Char /= Incoming_Msg.Terminator then
    Incoming_Msg.Append (Received_Char);
  end if;
  if Received_Char = Incoming_Msg.Terminator or else
    Incoming_Msg.Length = Incoming_Msg.Physical_Size
  then -- reception complete
    loop
      -- wait for device to clear the status
      exit when not Status (Device.Transceiver.all,
                           Read_Data_Register_Not_Empty);
    end loop;
    Disable_Interrupts (Device.Transceiver.all,
                       Source => Received_Data_Not_Empty);
    Incoming_Msg.Signal_Reception_Complete;
    Incoming_Msg := null;
  end if;
end Handle_Reception;
```

Note the call to *Signal_Reception_Complete* for the current *Message* object being received, designated by *Incoming_Msg*.

The alternative to a *Suspension_Object* is a parameterless protected entry that a task calls

to suspend itself. That certainly works in general, but we would need two entries, hence Jorvik. But also, the `Suspension_Object` approach can have a little better performance because it does not have the functionality that a protected entry has.

Note that type `Suspension_Object` might very well be implemented as a protected type. On a uniprocessor target, protected object mutual exclusion can be implemented via priorities, so it won't make much difference. (GNAT's bare-board run-times use that mutual exclusion implementation approach, as well as the PO implementation of type `Suspension_Object`.)

39.3 Pros

In all three idioms, the approach is directly expressed, meets the requirements, and hides the implementation details. The implementations are efficient relative to their requirements, the only reasonable metric. In particular, `Suspension_Objects` are expected to be faster than protected entries, but only support synchronization, and only with one caller at a time — there's no queue. Nor do they support communication. Protected entries have no such restrictions and are reasonably efficient given their considerable additional capabilities.

39.4 Cons

None.

39.5 Relationship With Other Idioms

The idiom showing how to connect a PO or task to an enclosing record object was illustrated by an interrupt handler PO, but that idiom is not necessary. Indeed, we used a protected type directly in the last implementation.

39.6 What About Priorities?

The idiom expressions do not determine the actual priorities assigned to the protected objects containing the handler procedures, nor those of the notified tasks.

The language standard requires the priorities for interrupt handler POs to be in the range defined by the subtype `System.Interrupt_Priority`. Under the Ravenscar and Jorvik profiles they must also satisfy the Ceiling Priority Protocol requirements.

The target's interrupt hardware may dictate the specific handler priorities, or at least their floor values. You may be able to control those hardware priorities via the target board startup code.

But usually we have some freedom to choose, so what priorities should be assigned?

Often the values are arbitrary. However, a more rigorous approach may be required. A good guideline is that if you need to do a timing (schedulability) analysis for the application tasks' deadlines, you need to do it for the interrupt handlers' deadlines too. The same analyses can be used, i.e., response-time analysis, and the same priority assignment schemes, i.e., a shorter period gets a higher priority. (The interrupt *period* is the minimum interval between the interrupt occurrences.)

In addition, ensuring interrupt handler deadlines are met is part of ensuring the tasks meet their deadlines. That's because the interrupt handlers release the associated sporadic (event-driven) tasks for execution. A sporadic task triggered by a device (say) usually

will have a deadline no greater than the next occurrence of the sensor-generated interrupt, that is, the interrupt period. The priority of the task will be set according to that period.

39.7 Notes

1. The *traditional* expression for an interrupt handler, i.e., a procedure, is allowed by the language as a vendor-defined extension. However, there will likely be language-oriented restrictions applied to those procedures, due to the context. That's true of other languages as well.
2. You shouldn't assign interrupt handler (PO) priorities by semantic importance, just as you shouldn't do so for task priorities. More *important* interrupt handlers shouldn't necessarily be assigned more urgent priorities.

REDUCING OBJECT CODE FROM GENERIC PACKAGE INSTANTIATIONS

40.1 Motivation

Generic unit instantiations are often, but not always, implemented by an Ada compiler as a form of *macro expansion*. In this approach, the compiler produces separate, dedicated object code for every instantiation. The macro expansion approach can produce better run-time performance but can result in large total object code size for the executable when there are many instances, especially when the generic packages instantiated contain a lot of unit declarations. For example, the generic I/O packages contained within package `Ada.Text_IO` are themselves relatively large.

The alternative compiler implementation approach is *code-sharing*, in which distinct instantiations of a given generic unit are implemented with shared object code in a single module.

Clearly, sharing the object code can reduce the total size, but code-sharing can be very complicated to implement, depending on the generic unit itself. For a trivial example, consider the following package:

```
generic
package P is
  Error : exception;
end P;
```

The semantics of the language require that every instantiation of generic package `P` be a distinct package, as if each instance was instead written explicitly as a non-generic package (at the point of instantiation) with the instance name. As a result, each package instance declares an exception, and these exceptions must be treated as distinct from each other. A code-sharing implementation must maintain that distinction with one object code module.

In the example above, there are no generic formal parameters, nor other declarations within the package declarative part besides the exception, because they are not necessary for that example. However, generic formal parameters can be a problem for code-sharing too. For example, consider this generic package:

```
generic
  Formal_Object : in out Integer;
package P is
  -- ...
end P;
```

This generic package has a generic formal object parameter with mode `in out`. (We chose type `Integer` purely for convenience.) That specific mode can cause a similar problem as seen in the exception example, because the mode allows the generic package to update the generic actual object passed to it. The shared object code must keep track of which object is updated during execution.

Therefore, when writing the application source code that instantiates generic packages, developers should do so in a manner that minimizes the amount of object code that might result.

40.2 Implementation(s)

The application source code should be written in a manner that shares the instantiations themselves, when possible, thereby reducing the number of instantiations that exist.

For example, let's say that several units in the application code require the ability to do I/O on some floating-point type. For simplicity, let's say that this is a type named `Real`, declared in a package named `Common`. Here is a declaration for an example package body that requires the I/O capability:

```
with Ada.Text_IO, Common;
package body User1 is
    package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
    -- ...
end User1;
```

That's certainly legal, and works, but we've said that several units require I/O for type `Real`. Let's say there are in fact twenty such units. They all do something similar:

```
with Ada.Text_IO, Common;
package body User20 is
    package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
    -- ...
end User20;
```

As a result, the application has twenty instantiations (at least) of `Ada.Text_IO.Float_IO`. There will be instances named `User1.Real_IO`, `User2.Real_IO`, and so on, up to `User20.Real_IO`. The fact that the local names are all `Real_IO` is irrelevant.

If the compiler happens to use the macro-expansion implementation, that means the application executable will have twenty copies of the object code defined by the generic `Float_IO`. For example, GNAT performs some internal restructuring to avoid this problem for these specific language-defined generic units, but not for application-defined generics.

Instead, we can simply instantiate the generic at the library level:

```
with Ada.Text_IO, Common;
package Real_IO is new Ada.Text_IO.Float_IO (Common.Real);
```

Because the instantiation occurred at the library level, the resulting instance is declared at the library level, and can therefore be named in a "with_clause" like any other library package.

```
with Real_IO;
package body User1 is
    -- ...
end User1;
```

Each client package can use the same instance via the with_clause, and there's only one instance so there's only one copy of the object code.

40.3 Pros

The total object code size is reduced, compared to the alternative of many local instantiations.

40.4 Cons

What would otherwise be an implementation detail hidden from clients can now become visible to them because a (public) library unit can be named in `with_clause` by any other unit. As a result, this approach should not be used in all cases, not even as a default design approach. Restricting the visibility of the instance may be more important than the amount of object code it contributes. Hiding implementation artifacts allows more freedom to change the implementation without requiring changes to client code.

40.5 Relationship With Other Idioms

None.

40.6 Notes

1. The reader should understand that this issue is not about the number of subprograms within any given package, whether or not the package is a generic package. In the past, some linkers included the entire object code for a given package (instance or not), regardless of the number of subprograms actually used from that package in the application code. That was an issue with reusable library code, for example packages providing mathematical functions. Modern linkers can be told not to include those subprograms not called by the application. For example, with `gcc`, the compiler can be told to put each subprogram in a separate section, and then the linker can be told to only include in the executable those sections actually referenced. (Data declarations can be reduced that way as well.)

RESOURCE ACQUISITION IS INITIALIZATION (RAII)

41.1 Motivation

In order for the expected semantics to be obtained, some types require clients to follow a specific protocol when calling the type's operations. Furthermore, failing to follow the protocol can cause system-wide ill effects.

For example, concurrency abstractions such as mutexes provide the mutually exclusive access necessary to prevent the race conditions that arise when competing concurrent threads access shared resources. These mutex objects must be 1) both acquired and released, 2) by every thread accessing that shared resource, 3) at the right places in the source code, and 4) in the proper order. Failure to acquire the mutex prior to accessing the shared resource leads to race conditions, and failure to release it can lead to deadlocks. Ensuring the mutex is released is complicated by the possibility of exceptions raised after the lock is acquired.

Although concurrency is a prime example, the issue is general in nature. We will continue with the concurrency context for the sake of discussion.

Like the classic *monitor* concept (³⁷², ³⁷³, ³⁷⁴) on which they are based, Ada defines a protected object (PO) as a concurrency construct that is higher-level and more robust than mutexes and semaphores. Those advantages accrue because the bodies of the protected operations are only responsible for implementing the functional requirements. The underlying run-time library is responsible for implementing the mutually exclusive access, and also thread management. As a result, the source code is much simpler and is robust even in the face of exceptions.

(In the works cited above, Hoare's contribution³⁷⁴ was equally important, but Hansen's contributions³⁷², ³⁷³ were reified in *Concurrent Pascal*³⁷⁰, a concrete programming language.)

However, a protected object is not always appropriate. Consider an existing sequential program that makes calls to visible procedures provided by a package:

```
package P is
  procedure Operation_1;
  procedure Operation_2;
  -- ...
end P;
```

Inside the package body are one or more state variables that are manipulated by the procedures (i.e., as in an *Abstract Data Machine* (page 1193)):

³⁷² P. B. Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.

³⁷³ P. B. Hansen. *Operating System Principles*. Prentice-Hall, 1973.

³⁷⁴ C. A. R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17(10):549-557, 1974.

³⁷⁰ https://en.wikipedia.org/wiki/Concurrent_Pascal

```
with Ada.Text_IO; use Ada.Text_IO;

package body P is

  State : Integer := 0;

  procedure Operation_1 is
  begin
    State := State + 1; -- for example...
    Put_Line ("State is now " & State'Image);
  end Operation_1;

  procedure Operation_2 is
  begin
    State := State - 1; -- for example...
    Put_Line ("State is now " & State'Image);
  end Operation_2;

  -- ...

end P;
```

This design is reasonable in a strictly sequential caller context. But if new application requirements are such that multiple tasks will be calling these procedures asynchronously, there is a problem. The package-level variable `State` will be subject to race conditions because it is (indirectly) shared among the calling tasks. Race conditions tend to be Heisenbugs because they are timing-dependent, so they can be exceedingly difficult to identify and expensive to debug.

In response to the new requirements, we could declare a protected object within the package body and move the declaration of `State` into that PO. In addition, we would declare two protected procedures corresponding to `Operation_1` and `Operation_2`. The two new protected procedure bodies would do what the original procedures did, including accessing and updating `State`. The original procedures — still presented to clients — would now call these new protected procedures:

```
with Ada.Text_IO; use Ada.Text_IO;

package body P is

  protected Threadsafe is
    procedure Operation_1;
    procedure Operation_2;
  private
    State : Integer := 0;
  end Threadsafe;

  protected body Threadsafe is

    procedure Operation_1 is
    begin
      State := State + 1; -- for example...
      Put_Line ("State is now " & State'Image);
    end Operation_1;

    procedure Operation_2 is
    begin
      State := State - 1; -- for example...
      Put_Line ("State is now " & State'Image);
    end Operation_2;

  end Threadsafe;

end P;
```

(continues on next page)

(continued from previous page)

```

end Threadsafe;

procedure Operation_1 is
begin
  Threadsafe.Operation_1;
end Operation_1;

procedure Operation_2 is
begin
  Threadsafe.Operation_2;
end Operation_2;

-- ...

end P;
```

As a result, there can be no race conditions on State, and the source code is both simple and robust.

We could put the PO in the package spec and then clients could call the protected object's operations directly, but changing all the clients could be expensive.

However, this new design is not portable because the two Threadsafe protected procedure bodies both call a potentially blocking operation, in this case `Ada.Text_IO.Put_Line`. Erroneous execution is the result. It might work as intended when executed, or it might do something else, or, if detected, `Program_Error` will be raised. On a run-time library built on top of an operating system, it may work as intended because the OS may provide thread locking mechanisms that the run-time library can use. In that case a blocking operation just suspends the caller thread's execution temporarily without releasing the PO lock. Although the blocking operation would allow some other caller task to be dispatched, no other caller could acquire that same PO lock, so race conditions are prevented within that PO. When the blocking operation returns, the protected procedure body can continue executing, still holding the lock. However, on a run-time library that does not use locks for mutual exclusion — it can use priorities, in particular — another caller to that same PO could access the enclosed variables while the first caller is blocked, thus breaking the mutually exclusive access guarantee.

Calling an I/O operation is not all that strange here, and those are not the only potentially blocking operation defined by the language.

Note that moving the calls to `Put_Line` out of the PO procedure bodies, back to the regular procedure bodies that call those PO procedures, would solve the portability problem but would not work functionally. There would be no guarantee that, during execution, the call to `Put_Line` would immediately follow the execution of the protected procedure called immediately before it in the source code. Hence the printed text might not reflect the current value of the State variable.

As a consequence, we must fall back to manually acquiring and releasing an explicit lock. For example, we could declare a lock object at the package level, as shown below, and have each operation acquire and release it:

```

with GNAT.Semaphores; use GNAT.Semaphores;
with Ada.Text_IO; use Ada.Text_IO;

package body P is

  subtype Mutual_Exclusion is Binary_Semaphore
    (Initially_Available => True,
     Ceiling => Default_Ceiling);

  Lock : Mutual_Exclusion;
```

(continues on next page)

(continued from previous page)

```

State : Integer := 0;

procedure Operation_1 is
begin
  Lock.Seize;
  State := State + 1; -- for example...
  Put_Line ("State is now" & State'Img);
  Lock.Release;
exception
  when others =>
    Lock.Release;
    raise;
end Operation_1;

procedure Operation_2 is
begin
  Lock.Seize;
  State := State - 1; -- for example...
  Put_Line ("State is now" & State'Img);
  Lock.Release;
exception
  when others =>
    Lock.Release;
    raise;
end Operation_2;

end P;

```

The subtype `Mutual_Exclusion` is just a binary semaphore with the discriminant set so that any object of the subtype is initially available. You can assume it is a protected type with classic binary semaphore semantics. See package `GNAT.Semaphores` for the details. The ceiling discriminant isn't important here, but we must set them all if we set any of them.

This design works, but the resulting code is clearly more complex and less robust than the PO approach.

41.2 Implementation

Our implementation uses an explicit global lock (a mutex), as above, but reintroduces automatic lock acquisition and release.

To achieve that automation, we leverage the language-defined object *lifetime* rules. These rules specify that an object is initialized when it is created and finalized when it is about to be destroyed. Initialization and finalization may be null operations, and thus absent from the object code, but application developers can define explicit initialization and finalization operations. When defined, these operations are called automatically by the underlying implementation, during the object's lifetime.

We will use the object initialization operation to seize the global lock and the object finalization operation to release it. The object lifetime rules will ensure that the lock's operations are called at the necessary times, thereby providing the required mutually exclusive access. In addition, the rules will ensure that the lock will be released even if an exception is raised in the bracketed application code.

Developers may be familiar with this approach under the name [Resource Acquisition Is Initialization](https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization)³⁷¹. Another name for this technique is *Scope-Bound Resource Management* because of the initialization and finalization steps invoked upon scope entry and exit.

³⁷¹ https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Therefore, we will create a new type with user-defined initialization and finalization operations. We name this new type `Lock_Manager` because the type provides a wrapper for locks, rather than being a lock directly. Object creation and destruction will invoke the initialization and finalization routines, automatically.

Because they are wrappers for locks, each object of this type will reference a distinct lock object so that the initialization and finalization operations can manipulate that lock object. We use an access discriminant to designate that lock. By doing so, we decouple the new type from the specific lock, and thus from the application code. Otherwise, the new facility would not be reusable.

The resulting relationship between the global shared lock and the local object will be as follows:

```
Lock : Mutual_Exclusion;

procedure Op is
  LM : Lock_Manager (<pointer to Lock>)
  -- <initialization automatically called for LM>
begin
  -- ... sequence of statements for Op
  -- <finalization called for LM>
end Op;
```

The language rules specify that a subprogram's local declarative part is elaborated prior to the execution of that subprogram's sequence of statements. During that elaboration, objects are created and initialized. The object creation for LM precedes the sequence of statements in the procedure body for Op, so the designated lock will be acquired prior to the shared resource use within that body.

Similarly, the rules specify that finalization occurs when an object is about to cease to exist, in this case because the local object LM goes out of scope. That won't happen until the end of the sequence of statements is reached for Op, in the normal case, so finalization will ensure that the lock is released after any possible reference in Op's statement sequence. The run-time will also invoke finalization in the face of exceptions because exceptions also cause the scope to be exited.

To define the `Lock_Manager` type, we declare it in a separate package as a tagged limited private type with a discriminant designating a `Mutual_Exclusion` object:

```
type Lock_Manager (Lock : not null access Mutual_Exclusion) is
  tagged limited private;
```

We make it a limited type because copying doesn't make sense semantically for `Lock_Manager` objects.

In addition, during optimization the compiler is allowed to remove unreferenced objects of non-limited types. As you saw above in procedure Op, there will be no explicit references to the object LM, so making the type limited prevents that unwanted optimization.

Only *controlled* types support user-defined initialization and finalization operations (as of Ada 2022). Therefore, in the package private part the type is fully declared as a controlled type derived from `Ada.Finalization.Limited_Controlled`, as shown below. We hide the fact that the type will be controlled because we don't intend `Initialize` and `Finalize` to be called manually by clients.

```
type Lock_Manager (Lock : not null access Mutual_Exclusion) is
  new Ada.Finalization.Limited_Controlled with null record;
```

No additional record components are required, beyond the access discriminant.

Immediately following the type declaration, we declare overriding versions of the inherited procedures `Initialize` and `Finalize`:

```
overriding procedure Initialize (This : in out Lock_Manager);
overriding procedure Finalize  (This : in out Lock_Manager);
```

These are the operations called automatically by the implementation.

The full package spec is as follows:

```
with Ada.Finalization;
with GNAT.Semaphores; use GNAT.Semaphores;

package Lock_Managers is

  subtype Mutual_Exclusion is Binary_Semaphore
    (Initially_Available => True,
     Ceiling             => Default_Ceiling);

  type Lock_Manager (Lock : not null access Mutual_Exclusion) is
    tagged limited private;

private

  type Lock_Manager (Lock : not null access Mutual_Exclusion) is
    new Ada.Finalization.Limited_Controlled with null record;

  overriding procedure Initialize (This : in out Lock_Manager);
  overriding procedure Finalize  (This : in out Lock_Manager);

end Lock_Managers;
```

The fact that there are no visible primitive operations tells the reader that this is a somewhat different *ADT* (page 1187). The most useful thing a client can do with such a type is to declare objects, but that's exactly what we want.

Each overridden procedure simply references the lock designated by the formal parameter's Lock discriminant:

```
package body Lock_Managers is

  -----
  -- Initialize --
  -----

  overriding procedure Initialize (This : in out Lock_Manager) is
  begin
    This.Lock.Seize;
  end Initialize;

  -----
  -- Finalize --
  -----

  overriding procedure Finalize (This : in out Lock_Manager) is
  begin
    This.Lock.Release;
  end Finalize;

end Lock_Managers;
```

The resulting user code is almost unchanged from the original sequential code:

```
with Ada.Text_IO;    use Ada.Text_IO;
with Lock_Managers;  use Lock_Managers;
```

(continues on next page)

(continued from previous page)

```

package body P is

  State : Integer := 0;

  Lock : aliased Mutual_Exclusion;

  procedure Operation_1 is
    LM : Lock_Manager (Lock'Access) with Unreferenced;
  begin
    State := State + 1; -- for example...
    Put_Line ("State is now" & State'Img);
  end Operation_1;

  procedure Operation_2 is
    LM : Lock_Manager (Lock'Access) with Unreferenced;
  begin
    State := State - 1; -- for example...
    Put_Line ("State is now" & State'Img);
  end Operation_2;

end P;

```

The aspect `Unreferenced` tells the compiler that no references in the source code are expected. That has two effects during compilation. First, warnings about the lack of references in the source code are disabled. Ordinarily we'd want those warnings because an unreferenced object usually indicates a coding error. That warning would be *noise* for objects of this type. But by the same token, the compiler will issue a warning if some explicit reference is present, perhaps added much later in the project lifetime.

41.3 Pros

Race conditions are precluded, the client code is simpler than direct manual calls, and the code is robust, especially concerning exceptions. These advantages are significant, given the cost in engineering time to debug the errors this design prevents.

41.4 Cons

The lock is global, so all calls go through it. Hence all calls are sequential, even if some could run concurrently. In the above example that's exactly as required, but in other situations it might be unnecessarily limiting.

Compared to the manual call approach, the run-time cost for keeping track of objects to be finalized could be non-trivial. That's likely true in any language.

41.5 Relationship With Other Idioms

None.

41.6 Notes

- The name for a similar type in the C++ Boost library is `Scoped_Lock`, as is the Ada type in the GNAT library package `GNATColl.Locks`. I used `Scope_Lock` in AdaCore's

Gem #70³⁷⁵.

- I didn't invent the name `Scope_Lock` or the Ada implementation, but I don't recall where I first saw it many years ago. My apologies to that author.
- I consider the name `Lock_Manager` or something similar to be better, since objects of the type are wrappers for locks, not locks themselves. Indeed, in C++ 2011 the name is `lock_guard`.

³⁷⁵ P. Rogers. *Gem #70: The Scope Locks Idiom*. <https://www.adacore.com/gems/gem-70>, 2009.

USING STREAMS FOR API FLEXIBILITY

42.1 Motivation

Software interfaces for hardware devices usually support only primitive data types. Communications devices such as serial I/O ports and network adapters are good examples. Their device drivers provide an API for sending and receiving individual 8-bit or 9-bit numeric quantities, or sequences of these. Software clients — either the application, or higher-level layers of device interfaces — may want to send and receive more complex data types. If so, how can the device driver support them?

We certainly don't want the clients to do unchecked conversions everywhere. That's error-prone, it prevents the compiler from checking the usage, and makes clients responsible for what should be an internal implementation detail.

This is a general issue, not specific to communications hardware, but we will discuss it in that context because of the familiarity of such devices.

The traditional approach is for the device driver's I/O routines to have two parameters for this purpose: an address and a length. The address indicates the first byte of the client value to be sent or received, and the length indicates how many bytes are involved. Values of any type can be sent or received using this interface, but it's a very unsafe / unrobust approach. Developers could pass the wrong starting address, or specify the wrong length, thus potentially transmitting only part of the intended value or including part of some wholly unrelated object. Moreover, developers could pass the address and length of some object that is not of the type expected on the other end of the connection. After all, Ada allows us to take the address of just about anything. The effect on the receiver would be difficult to predict. These mistakes are very expensive to locate, and the compiler cannot help.

We need a type-safe approach for sending and receiving higher-level types so that the compiler can catch our mistakes. After all, preventing coding errors is much cheaper than fixing them later.

42.2 Implementation

We will explore the possibilities using a concrete `USART` (Universal Synchronous/Asynchronous Receiver Transmitter)³⁷⁶ defined in the `Ada Drivers Library` (ADL)³⁷⁷. A USART is the physical communications device underlying what is commonly referred to as a *serial port*. That name reflects the fact that the device transmits and receives data serially, as opposed to in parallel.

The ADL provides packages representing specific microcontrollers as well as their on-chip peripherals, including USARTs, timers, DMA controllers, and so forth. Each kind of peripheral is represented by a dedicated *Abstract Data Type (ADT)* (page 1187). The following is the elided ADT declaration for USARTs on STM32 microcontrollers:

³⁷⁶ https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

³⁷⁷ https://github.com/AdaCore/Ada_Drivers_Library

```

-- ...

package STM32.USARTs is

    type USART is ... private;

    -- ...
    procedure Receive (This : USART;          Data : out UInt9);
    procedure Transmit (This : in out USART;   Data : UInt9);

    function Tx_Ready (This : USART) return Boolean;
    function Rx_Ready (This : USART) return Boolean;

    -- ...

private
    -- ...
end STM32.USARTs;

```

Note the formal parameter data type for both `Receive` and `Transmit`, i.e., a single 9-bit unsigned numeric value. Although a client might actually want to send and receive such values directly, that's probably not the case.

Our implementation is structured as a package hierarchy rooted at package `Serial_IO`. (This implementation is part of an example in the ADL — see [Note #1](#) (page 1296) below.) This root package declares a record type and routines that are common to any implementation. The type, named `Peripheral_Descriptor`, contains a component named `Transceiver` that will designate the actual on-chip USART device being driven. The other record components are required for connecting that device to the external world.

Type `Peripheral_Descriptor` is hardware-specific and is therefore not defined as an ADT. Instead, the package uses the *Groups of Related Program Units* (page 1184) idiom.

```

with STM32;          use STM32;
with STM32.GPIO;     use STM32.GPIO;
with STM32.USARTs;   use STM32.USARTs;

package Serial_IO is

    type Peripheral_Descriptor is record
        Transceiver      : not null access USART;
        Transceiver_AF   : GPIO_Alternate_Function;
        Tx_Pin           : GPIO_Point;
        Rx_Pin           : GPIO_Point;
    end record;

    procedure Initialize_Hardware (Device : Peripheral_Descriptor);
    -- enable clocks, configure GPIO pins, etc.

    procedure Configure
        (Device      : access USART;
         Baud_Rate   : Baud_Rates;
         Parity      : Parities      := No_Parity;
         Data_Bits   : Word_Lengths  := Word_Length_8;
         End_Bits    : Stop_Bits     := Stopbits_1;
         Control     : Flow_Control  := No_Flow_Control);

    -- ...

end Serial_IO;

```

Procedure `Configure` is a convenience routine, provided because a specific sequence of

driver calls is required in order to set the individual parameters.

Clients will call these procedures directly to set up the STM32 on-chip USART device.

Our implementation will consist of an ADT named `Serial_Port`, and a means for sending and receiving values of higher-level types via `Serial_Port` objects. Type `Serial_Port` will be a wrapper for the device driver's USART type. Therefore, the type `Serial_Port` will have an access discriminant designating a USART:

```
type Serial_Port (Device : not null access USART) is ...
```

Using this Device discriminant a `Serial_Port` object can reference the wrapped physical USART in order to send and receive values via that hardware device.

With that introduction in place, we can consider the possible approaches to sending and receiving values of higher-level types via `Serial_Port` objects.

The canonical Ada approach consists of a generic package with a generic formal parameter type. That formal type represents a client-specific type to be sent or received. Clients instantiate the generic for every client-defined type necessary.

The generic package would look like the following:

```
generic
  type Client_Data (<>) is limited private;
package Client_IO is

  procedure Send
    (This      : in out Serial_Port;
     Outgoing  : Client_Data);

  procedure Receive
    (This      : in out Serial_Port;
     Incoming  : out Client_Data);

end Client_IO;
```

In the procedure bodies, the values of the Incoming or Outgoing parameters would be converted to or from bytes as necessary and sent or received via the USART designated by `This.Device`.

This approach supports as many client-level types as required, including limited types. It is type-safe so the compiler can catch errors in the type(s) being sent and received. In addition, the low-level implementation details, such as unchecked conversions, are hidden inside the generic package body.

Moreover, the approach is independent of other design considerations, such as whether callers wait for completion of the invoked I/O operation. The bodies of the generic procedures can be implemented to provide the expected behavior.

However, this approach is somewhat heavy because the generic package must be instantiated for every client type to be supported. If there are many such types, there will be many instantiations. Not only is that more lines of source code, but also more object code because most Ada implementations do not support code-sharing for generic instantiations. But that said, the need for a given client to send and receive values of many different types is not typical.

However, there is a more concise approach possible, for both the driver and client source code. This alternative approach leverages the flexibility of streams and stream attributes. Using streams allows the wrapper to support an unlimited number of distinct client types, with no additional source code required per type.

Recall that the stream attributes are callable routines whose first parameter is an access value designating some stream object. The formal parameter type is access-to-class-wide,

so any stream object is allowed. For example, the notional specification for the `'Output` attribute looks like this, for some subtype `S` of type `T`:

```
procedure S'Output
  (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T);
```

Therefore, the fundamental approach will be to declare the `Serial_Port` ADT as a stream type. Clients can then send and receive values simply by invoking the stream attributes, passing (access to) `Serial_Port` objects as the first parameter to the invocations.

The `Serial_Port` ADT will be defined in package `Serial_IO.Streaming`. Given that, a client can declare a `Serial_Port` object like so:

```
with STM32.Device;
with Serial_IO.Streaming; use Serial_IO;

package Peripherals_Streaming is

  -- the USART selection is arbitrary but the AF number and the
  -- pins must be those required by that USART
  Peripheral : constant Serial_IO.Peripheral_Descriptor :=
    (Transceiver => STM32.Device.USART_1'Access,
     Transceiver_AF => STM32.Device.GPIO_AF_USART1_7,
     Tx_Pin       => STM32.Device.PB6,
     Rx_Pin       => STM32.Device.PB7);

  COM : aliased Streaming.Serial_Port (Peripheral.Transceiver);

end Peripherals_Streaming;
```

In the above, `Peripheral` is an object that describes a specific USART on the STM32 Discovery Board microcontroller, along with the values necessary to connect that specific USART to the external environment. The `Peripheral` variable will be passed to a call to `Initialize_Hardware`. Similarly, `COM` is an object that wraps the USART designated by `Peripheral`.

Because `Serial_Port` will be a stream type (it will be in the derivation class rooted at `Root_Stream_Type`), `COM` will be a streaming object that we can pass to invocations of the stream attributes. For example, to send a **String** value via `COM` we could write:

```
String'Output (COM'Access, "Hello World");
```

To send an **Integer** value:

```
Integer'Write (COM'Access, 42);
```

To receive an **Integer** value into the **Integer** object `X`:

```
Integer'Read (COM'Access, X);
```

That's all clients must do to send and receive values via the USART wrapped by `COM`. They could do the same for floating-point types, record types, and so on. Objects of any type with the streaming attributes defined can be sent or received, and in any order.

To make `Serial_Port` a stream type, the declaration visibly extends `Ada.Streams.Root_Stream_Type`:

```
type Serial_Port (Device : not null access USART) is
  new Ada.Streams.Root_Stream_Type with private;
```

This is *Interface Inheritance* (page 1237) so that clients can treat `Serial_Port` as a stream type. The private extension hides implementation details that we'll describe momentarily.

As a concrete extension of `Root_Stream_Type` we must declare overridings for procedures `Read` and `Write`:

```

overriding
procedure Read
  (This   : in out Serial_Port;
   Buffer  : out Ada.Streams.Stream_Element_Array;
   Last   : out Ada.Streams.Stream_Element_Offset);

overriding
procedure Write
  (This   : in out Serial_Port;
   Buffer  : Ada.Streams.Stream_Element_Array);

```

`Stream_Element_Array` is an unconstrained array type with `Stream_Element` as the array component. `Stream_Element` is an unsigned numeric type corresponding to a machine storage element, e.g., a byte.

Procedure `Write` inserts these array components into the designated stream. Procedure `Read` consumes the array components from the stream and includes a parameter indicating the index of the last component assigned.

These two procedures are called by the various stream attributes' implementations, not by clients. For example, consider again the call to `'Output`:

```
String'Output (COM'Access, "Hello World");
```

The call dispatches to our overriding of procedure `Write` because the first parameter designates our specific stream object.

For both procedures the array components hold the serialized representation of the value read from, or to be written to, the stream. For procedure `Write`, the array contains the stream-oriented representation of the client value, e.g., the "Hello World!" passed to `String'Output`. For procedure `Read`, the array contains the value consumed from the stream that will be converted into the client type, e.g., type `Integer` for a call to `Integer'Read`, and loaded into the client variable.

Note that the two procedures are only responsible for reading or writing the array components from/to the specified stream. Conversions between the types in the clients' attribute invocations and type `Stream_Element_Array` are not their responsibility. That metamorphosis is handled automatically by the language-defined attributes' implementations. It's a nice separation of concerns.

Here then is the full package declaration for the `Serial_Port` ADT:

```

with Ada.Streams;
with Ada.Real_Time; use Ada.Real_Time;

package Serial_IO.Streaming is
  pragma Elaborate_Body;

  type Serial_Port (Device : not null access USART) is
    new Ada.Streams.Root_Stream_Type with private;

  procedure Set_Read_Timeout
    (This : in out Serial_Port;
     Wait : Time_Span);
  -- Stream attributes that call Read (below) can either wait
  -- indefinitely or can be set to return any current values
  -- received after a given interval. If the value Time_Span_Last
  -- is passed to Wait, the effect is essentially to wait forever,
  -- i.e., blocking. That is also the effect if this routine is
  -- never called.

```

(continues on next page)

(continued from previous page)

```

overriding
procedure Read
  (This   : in out Serial_Port;
   Buffer  : out Ada.Streams.Stream_Element_Array;
   Last   : out Ada.Streams.Stream_Element_Offset);

overriding
procedure Write
  (This   : in out Serial_Port;
   Buffer  : Ada.Streams.Stream_Element_Array);

private

type Serial_Port (Device : access USART) is
  new Ada.Streams.Root_Stream_Type with record
    Timeout : Time_Span := Time_Span_Last;
  end record;

procedure Await_Send_Ready (This : access USART) with Inline;

procedure Await_Data_Available
  (This       : access USART;
   Timeout    : Time_Span := Time_Span_Last;
   Timed_Out  : out Boolean)
with Inline;

use Ada.Streams;

function Last_Index
  (First : Stream_Element_Offset;
   Count : Long_Integer)
  return Stream_Element_Offset
with Inline;

end Serial_IO.Streaming;

```

Prior to procedures Read and Write, the package declares a procedure for controlling a timeout associated with a Serial_Port stream. This timeout controls how long procedure Read should wait for input to be available in the stream. The default is to wait for what amounts to forever. Note that the timeout applies both to the case of some input received, and none received.

In the package private part we see that the type extension contains the record component named Timeout, with the initial value providing the default. That's the only other record component required, besides the discriminant.

Additional implementation-oriented routines are also declared there, rather than in the package body, for the sake of any child packages that might be declared in the future. Note in particular the two that await I/O completion, as this is a blocking implementation.

Here is the package body:

```

with HAL;

package body Serial_IO.Streaming is

  -----
  -- Set_Read_Timeout --
  -----

```

(continues on next page)

(continued from previous page)

```

procedure Set_Read_Timeout
  (This : in out Serial_Port;
   Wait : Time_Span)
is
begin
  This.Timeout := Wait;
end Set_Read_Timeout;

-----
-- Await_Send_Ready --
-----

procedure Await_Send_Ready (This : access USART) is
begin
  loop
    exit when This.Tx_Ready;
  end loop;
end Await_Send_Ready;

-----
-- Await_Data_Available --
-----

procedure Await_Data_Available
  (This      : access USART;
   Timeout   : Time_Span := Time_Span_Last;
   Timed_Out : out Boolean)
is
  Deadline : constant Time := Clock + Timeout;
begin
  Timed_Out := True;
  while Clock < Deadline loop
    if This.Rx_Ready then
      Timed_Out := False;
      exit;
    end if;
  end loop;
end Await_Data_Available;

-----
-- Last_Index --
-----

function Last_Index
  (First : Stream_Element_Offset;
   Count : Long_Integer)
  return Stream_Element_Offset
is
begin
  if First = Stream_Element_Offset'First and then Count = 0 then
    -- although we intend to return First - 1, we cannot
    raise Constraint_Error; -- per AI95-227
  else
    return First + Stream_Element_Offset (Count) - 1;
  end if;
end Last_Index;

-----
-- Read --
-----

```

(continues on next page)

(continued from previous page)

```

overriding
procedure Read
  (This : in out Serial_Port;
   Buffer : out Ada.Streams.Stream_Element_Array;
   Last : out Ada.Streams.Stream_Element_Offset)
is
  Raw      : HAL.UInt9;
  Timed_Out : Boolean;
  Count    : Long_Integer := 0;
begin
  Receiving : for K in Buffer'Range loop
    Await_Data_Available (This.Device, This.Timeout, Timed_Out);
    exit Receiving when Timed_Out;
    This.Device.Receive (Raw);
    Buffer (K) := Stream_Element (Raw);
    Count := Count + 1;
  end loop Receiving;
  Last := Last_Index (Buffer'First, Count);
end Read;

-----
-- Write --
-----

overriding
procedure Write
  (This : in out Serial_Port;
   Buffer : Ada.Streams.Stream_Element_Array)
is
begin
  for Next of Buffer loop
    Await_Send_Ready (This.Device);
    This.Device.Transmit (HAL.UInt9 (Next));
  end loop;
end Write;

end Serial_IO.Streaming;

```

Procedure Read polls the wrapped USART device, continually, until a byte becomes available or the timeout is reached. Procedure Await_Data_Available performs this timed polling. Polling without relinquishing the processor is extremely questionable on a main CPU, but in a device driver on a dedicated microcontroller it is not necessarily a poor choice. But if polling is a problem, there is nothing preventing a non-blocking, interrupt-based implementation with a stream-based client API.

The function Last_Index is a convenience function called by procedure Read. It is used to compute Read.Last, the index of the last array component assigned in Read.Buffer. The function result is Buffer'First - 1 when no components are assigned, except when that would be less than the lowest possible array index value.

Here is a demonstration procedure to be run on the STM32 F4 Discovery Board:

```

with Last_Chance_Handler; pragma Unreferenced (Last_Chance_Handler);
with Serial_IO;
with Peripherals_Sstreaming; use Peripherals_Sstreaming;

procedure Demo_Serial_Port_Sstreaming is
begin
  Serial_IO.Initialize_Hardware (Peripheral);
  Serial_IO.Configure (COM.Device, Baud_Rate => 115_200);
  -- This baud rate selection is entirely arbitrary. Note that you may

```

(continues on next page)

(continued from previous page)

```

-- have to alter the settings of your host serial port to match this
-- baud rate, or just change the above to match whatever the host
-- serial port has set already. An application such as TerraTerm
-- or RealTerm is helpful.

loop
  declare
    -- await the next msg from the serial port
    Incoming : constant String := String'Input (COM'Access);
  begin
    -- echo the received msg content
    String'Output (COM'Access, "You sent '" & Incoming & "'");
  end;
end loop;
end Demo_Serial_Port_Streaming;

```

The specific USART on the STM32 F4 Discovery Board must be connected to a serial port on the host computer. With that connection in place the embedded ARM board and the host computer can communicate over the two serial ports. This demonstration iteratively receives a string sent from the host, prepends some text, and sends that back, in effect echoing the host sender's text.

The stream attributes `String'Output` and `String'Input` write and read the bounds as well as the characters. As a consequence, you will need to use a program on the host that handles those bounds. A good way to do that is to use a host program that also uses streams to send and receive `String` values. Note that the ADL serial port examples include a host application that you can build and run for this purpose.

42.3 Pros

The stream-based approach has all the advantages of the generic-based approach without requiring generic instantiations. There is no limit to the number and kinds of client types supported, including limited types if they have the attributes defined. It is type-safe by default, because the compiler will verify that the type used to invoke a stream attribute is the same type as the value involved. In addition, the low-level implementation details are hidden inside the package body.

Furthermore, the approach is independent of other design considerations, such as whether callers wait for completion of the invoked I/O operation.

Because it is maximally flexible and concise, we consider it the best implementation for this idiom. The generic-based approach remains a good one, however.

42.4 Cons

Limited types do not support the stream I/O attributes by default, but developers can define them. Note that this is not a problem for the generic-based approach, because we declared the generic formal type as `limited` and wouldn't need to do anything within the generic that would contradict that. The client's generic actual type can then be either a limited type or not.

When multiple types are being sent and received, the sender and receiver must be coordinated so that the next value consumed from the stream is of the type expected by the receiver. For example, the next value in the stream might have been written by the sender as a floating-point value, via `Float'Write (...)`. The receiver must use `Float'Read(...)` to consume that value from the stream. Arguably, this is not really a *con* because it's true for any stream when multiple types are involved. Even if we used the generic-based

approach, developers could instantiate the generic multiple times with different types and send their values via the same port. With streams this approach is as type-safe as it can be. However, see [Note #2](#) (page 1296) below for a possible mitigation.

42.5 Relationship With Other Idioms

As stated above, we use [Interface Inheritance](#) (page 1237) to visibly extend the root stream type.

42.6 Notes

1. You can find the streams-based approach, and others, in the *serial_ports* example in the ADL, for an STM32 F4 Discovery Board, found in [Ada_Drivers_Library/examples/STM32F4_DISCO/*](#)³⁷⁸. You can build and run them using the GNAT project file named [serial_ports_f4disco.gpr](#)³⁷⁹ located in that directory. See the [Ada_Drivers_Library/examples/shared/serial_ports/README.md](#)³⁸⁰ file for how to run them, including the special cable required for connecting the target board to the host computer. Note that a non-blocking, interrupt-drive approach is included there, although it is not stream-based.
2. In the [Cons section](#) (page 1295) above, we mentioned the coordination issue that arises when values of multiple types are inserted and retrieved from a given stream. A possible alternative would be to send and receive only tagged types in a given class hierarchy. The receiver could then use the language-defined [Generic_Dispatching_Constructor](#) to dynamically dispatch to constructors for the values received from the stream. Thus, the receiver would not need to know in advance what specific types of values are incoming.

³⁷⁸ https://github.com/AdaCore/Ada_Drivers_Library/tree/master/examples/STM32F4_DISCO

³⁷⁹ https://github.com/AdaCore/Ada_Drivers_Library/blob/master/examples/STM32F4_DISCO/serial_ports_f4disco.gpr

³⁸⁰ https://github.com/AdaCore/Ada_Drivers_Library/tree/master/examples/shared

DEALING WITH SILENT TASK TERMINATION

43.1 Motivation

A task completes abnormally when an exception is raised in its sequence of statements and is not handled. Even if the task body has a matching exception handler and it executes, the task still completes after the handler executes, although this time it completes normally. Similarly, if a task is aborted the task completes, again abnormally.

Whatever the cause, once completed a task will (eventually) terminate, and it does this silently — there is no notification or logging of the termination to the external environment. A vendor could support notification via their run-time library³⁸¹, but the language standard does not require it and most vendors — if not all — do not.

Nevertheless, applications may require some sort of notification of the event that caused the termination. Assuming the developer is responsible for implementing it, how can the requirement best be met?

43.2 Implementation

For unhandled exceptions, the simplest approach to silent termination is to define the announcement or logging response as an exception handler located in the task body exception handler part:

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
-- ...
task body Worker is
begin
  -- ...
exception
  when Error : others => -- last wishes
    Put_Line ("Task Worker terminated due to " & Exception_Name (Error));
end Worker;
```

A handler at this level expresses the task's *last wishes* prior to completion, in this case printing the names of the task and the active exception to `Standard_Output`. (We could print the associated exception message too, if desired.) The **others** choice covers all exceptions not previously covered, so in the above it covers all exceptions. Specific exceptions also could be covered, but the **others** choice should be included (at the end) to ensure no exception occurrence can be missed.

You'll probably want this sort of handler for every application task if you want it for any of them. That's somewhat inconvenient if there are many tasks in the application, but feasible. Possible mitigation includes the use of a task type. In that case you need only define the handler once, in the task type's body. You could even declare such a task type

³⁸¹ The Verdex Ada Development System did so.

inside a generic package, with generic formal subprograms for the normal processing and the exception handler's processing. That would make the task type reusable. But that's a bit heavy, and it can be awkward. See the [Notes](#) (page 1309) section below for details.

Alternatively, we could prevent unhandled exceptions from causing termination in the first place. We can do that by preventing task completion, via some additional constructs that prevent reaching the end of the task's sequence of statements. We will show these constructs incrementally for the sake of clarity.

Before we do, note that many tasks are intended to run until power is removed, so they have an infinite loop at the outer level as illustrated in the code below. For the sake of clarity and realism, we name the loop `Normal` and call some procedures to show the typical structure, along with the last wishes handler:

```
task body Worker is
begin
    Initialize_State;
    Normal : loop
        Do_Actual_Work;
    end loop Normal;
exception
    when Error : others => -- last wishes
        Put_Line ("Task Worker terminated due to " &
            Exception_Name (Error));
end Worker;
```

In the above, the procedures' names indicate what is done at that point in the code. The steps performed may or may not be done by actual procedure calls.

Strictly speaking, the optional exception handler part of the task body is the very end of the task's sequence of statements (the *handled sequence of statements*). We want to prevent the thread of control reaching that end — which would happen if any handler there ever executed — because the task would then complete.

Therefore, we first wrap the existing code inside a block statement. The task body's exception handler section becomes part of the block statement rather than at the top level of the task:

```
task body Worker is
begin
    begin
        Initialize_State;
        Normal : loop
            Do_Actual_Work;
        end loop Normal;
    exception
        when Error : others => -- last wishes
            Put_Line ("Task Worker terminated due to " &
                Exception_Name (Error));
    end;
end Worker;
```

Now any exception raised within `Initialize_State` and `Do_Actual_Work` will be caught in the block statement's handler, not the final part of the task's sequence of statements. Nothing else changes, semantically. The task will still complete because the block statement exits after the handler executes, and so far there's nothing after that block statement. We need to make one more addition.

The second (and final) addition prevents reaching the end of the sequence of statements after a handler executes, and hence the task from completing. This is accomplished by wrapping the new block statement inside a new loop statement. We name this outermost loop `Recovery`:

```

task body Worker is
begin
  Recovery : loop
  begin
    Initialize_State;
    Normal : loop
      Do_Actual_Work;
    end loop Normal;
  exception
    when Error : others =>
      Put_Line (Exception_Name (Error) & " handled in task Worker");
  end;
end loop Recovery;
end Worker;

```

When the block exits after the exception handler prints the announcement, normal execution resumes and the end of the Recovery loop is reached. The thread of control then continues at the top of the loop. Of course, absent an unhandled exception reaching this level, the Normal loop is never exited in the first place.

These two additions ensure that — with one caveat — Worker never terminates due to an unhandled exception raised during execution of the task's sequence of statements.

Note that an exception raised during elaboration of the task body's declarative part is not handled by the approach, or any other approach at this level, because the exception is propagated immediately to the master of the task. Such a task never reaches the handled sequence of statements in the first place.

The caveat concerns the language-defined exception `Storage_Error`. This exception requires special consideration because the reasons for raising it include exhausting the storage required for execution itself.

There are a couple of scenarios to consider.

The first scenario is task activation, i.e., creation. Initial task activation involves execution (in the tasking part of the run-time library) before the sequence of steps is reached. Hence task activation for the Worker task could fail due to an insufficient initial storage allocation. But because that failure happens before the block statement is entered it doesn't really apply to the caveat above.

The second scenario involves execution within the task's actual sequence of statements. Therefore it does apply to the caveat above. Here's why.

When called, the execution of a given subprogram requires a representation in storage, often known as a *frame*. Because subprogram calls and their returns can be seen as a series of stack pushes and pops, the representation for execution is typically via a stack of these frames. Calls cause stack frame pushes, creating new frames on the stack, and returns cause stack pops, reclaiming the frames. On exit, execution returns to the caller, so the previous top of the stack is now the active frame. Representation as a stack of frames works well so it is very common. (Functions returning values of unconstrained types are problematic because the size of the result isn't known at the point of the call, so the required frame size isn't known when the push occurs. Solutions vary, but that's a topic for another day.)

Now, suppose the task's sequence of statements includes a long series of subprogram calls, in which one subprogram calls another, and that one calls another, and so on, and none of these calls has yet returned. Eventually, of course, the dynamic call chain will end because the calls will return, at least in normal code. But let's suppose that the call chain is long and the most recent call has not yet returned to the caller.

In that case it is possible for one more call to exhaust the storage available for that task's execution. You can easily construct such a chain by calling an infinitely recursive procedure or function:

```
procedure P;  
  
procedure P is  
begin  
  P;  
end P;
```

When executing on a host OS it might take a very long time for a call to P to exhaust available storage, maybe longer than you'd be willing to wait. But on an embedded system, where physical storage is limited and there's no virtual memory, it might not take long at all.

Now, you might think that you don't use recursion, much less infinitely recursive routines, so this problem doesn't apply to you. But recursion is just an easy illustration. How long a call chain is too long? It depends on the memory resources available.

Moreover, exhaustion is not due only to the storage required for the call/return semantics. Frames include the representation of the local objects declared within the subprograms' declarative parts, if any.

```
procedure P;  
  
procedure P is  
  Local : Integer;  
begin  
  Local := 0;  
  P;  
end P;
```

Each execution of a call to P creates a semantically distinct instance of Local. A new frame containing the storage for each call's copy of Local implements that requirement nicely.

Of course, different subprograms usually declare different local objects, if they declare any at all. Because the storage required for these declarations varies, the corresponding frame sizes vary.

We can use that fact to reduce the length of the dynamic call chain required to illustrate storage exhaustion. The called subprograms will declare very large objects within their declarative parts. Hence each frame is correspondingly larger than if the subprogram declared nothing locally. Continuing the infinitely recursive subprogram example:

```
procedure P;  
  
procedure P is  
  type Huge_Component is array (Long_Long_Integer) of Long_Float;  
  type Huge_Array is array (Long_Long_Integer) of Huge_Component;  
  Local : Huge_Array;  
begin  
  Local := (others => (others => 0.0));  
  P;  
end P;
```

The size of the frame for an individual call to P will be very large indeed, if it is representable at all. Fewer calls will be required before Storage_Error is raised.

Now, with all that said, let's get back to this approach to silent termination. Here's the code again:

```
task body Worker is  
begin  
  Recovery : loop  
    begin  
      Initialize_State;
```

(continues on next page)

(continued from previous page)

```

    Normal : loop
        Do_Actual_Work;
    end loop Normal;
exception
    when Error : others =>
        Put_Line (Exception_Name (Error) & " handled in task Worker.");
    end;
end loop Recovery;
end Worker;

```

At this point you might be thinking that `Storage_Error` would be caught by the **others** choice anyway, so this (long-winded) talk about stack frames and dynamic call chains is irrelevant. That's where the caveat comes into play.

Specifically, if there's insufficient storage remaining for execution to continue, how do we know there's enough storage remaining to execute the exception handler? For that matter, how do we even know there's enough storage available for the run-time library to find the handler in the first place? Absent a storage analysis, we can't know with certainty.

Therefore, if the application matters, perform a worst-case storage analysis per task, including the exception handlers, and explicitly specify the tasks' stacks accordingly. For example:

```
task Worker with Storage_Size => System_Config.Worker_Storage;
```

We've defined the value as a constant named `Worker_Storage` declared in an application-defined package `System_Config`. All such values are declared in that package, for the sake of centralizing all the application's configuration parameters. We'd declare all the tasks' priorities there too.

Finally, although this approach works, the state initialization requires some thought.

As shown above, full initialization is performed again when the `Recovery` loop circles back around to the top of the loop. As a result, the *normal* processing in `Do_Actual_Work` must be prepared for suddenly encountering completely different state, i.e., a restart to the initial state. If that is not feasible the call to `Initialize_State` could be moved outside, prior to the start of the `Recovery` loop, so that it only executes once. Perhaps a different initialization procedure could be called after the exception handler to do partial initialization. Whether or not that will suffice depends on the application.

However, these approaches do not address task termination due to task abort statements.

Aborting tasks is both messy and expensive at run-time. If a task is updating some object and is aborted before it finishes the update, that object is potentially corrupted. That's the messiness. If an aborted task has dependent tasks, all the dependents are aborted too, transitively. A task in a rendezvous with the aborted task is affected, as are those queued waiting to rendezvous with it, and so on. That's part of the expensiveness when aborts are actually used. Worse, even if never used, abort statements impose an expense at run-time. The language semantics requires checks for an aborted task at certain places within the run-time library. Those checks are executed even if no task abort statement is ever used in the application. To avoid that distributed cost, you would need to apply a tasking profile disallowing abort statements and build the executable with a correspondingly reduced run-time library implementation.

As a consequence, aborting a task should be very rarely done. Regardless, the task abort statement exists. How can we express a *last wishes* response for that cause too?

Fortunately, Ada provides a facility that addresses all possible causes: normal termination, termination due to task abort, and termination due to unhandled exceptions.

With this facility developers specify procedures that are invoked automatically by the run-time library during task finalization. These procedures express the last wishes for the task,

but do not require any source code within the task, unlike the exception handler in each task body described earlier. These response procedures are known as *handlers*.

During execution, handlers can be applied to an individual task or to groups of related tasks. Handlers can also be removed from those tasks or replaced with other handlers. Because procedures are not first-class entities in Ada, handlers are assigned and removed by passing access values designating them.

The facility is defined by package `Ada.Task_Termination`. The package declaration for this language-defined facility follows, with slight changes for easier comprehension.

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Exceptions;         use Ada.Exceptions;

package Ada.Task_Termination
-- ...
is
  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

  type Termination_Handler is access protected procedure
    (Cause : in Cause_Of_Termination;
     T      : in Task_Id;
     X      : in Exception_Occurrence);

  procedure Set_Dependents_Fallback_Handler
    (Handler : in Termination_Handler);

  function Current_Task_Fallback_Handler
    return Termination_Handler;

  procedure Set_Specific_Handler
    (T      : in Task_Id;
     Handler : in Termination_Handler);

  function Specific_Handler (T : Task_Id) return Termination_Handler;
end Ada.Task_Termination;
```

As shown, termination handlers are actually protected procedures, with a specific parameter profile. Therefore, the type `Termination_Handler` is an access-to-protected-procedure with that signature. The compiler ensures that any designated protected procedure matches the parameter profile.

Termination handlers apply either to a specific task or to a group of related tasks, including potentially all tasks in the partition. Each task has one, both, or neither kind of handler. By default none apply. (Unless a partition is part of a distributed program, a single partition constitutes an entire Ada program.)

Clients call procedure `Set_Specific_Handler` to apply the protected procedure designated by `Handler` to the task with the specific `Task_Id` value `T`. These are known as *specific* handlers. The use of a `Task_Id` to specify the task, rather than the task name, means that we can set or remove a handler without having direct visibility to the task in question.

Clients call procedure `Set_Dependents_Fallback_Handler` to apply the protected procedure designated by `Handler` to the task making the call, i.e., the current task, and to all tasks that are dependents of that task. These handlers are known as *fall-back* handlers.

Handlers are invoked automatically, with the following semantics:

1. If a specific handler is set for the terminating task, it is called and then the response finishes.
2. If no specific handler is set for the terminating task, the run-time library searches for a fall-back handler. The search is recursive, up the hierarchy of task masters, including, ultimately, the environment task. If no fall-back handler is found no handler calls are

made whatsoever. If a fall-back handler is found it is called and then the response finishes; no further searching or handler calls occur.

As a result, at most one handler is called in response to any given task termination.

The following client package illustrates the approach. Package `Obituary` declares protected object `Obituary.Writer`, which declares two protected procedures. Both match the profile specified by type `Termination_Handler`. One such procedure would suffice, we just provide two for the sake of illustrating the flexibility of the dynamic approach.

Listing 1751: `obituary.ads`

```

1  with Ada.Exceptions;           use Ada.Exceptions;
2  with Ada.Task_Termination;     use Ada.Task_Termination;
3  with Ada.Task_Identification;  use Ada.Task_Identification;
4
5  package Obituary is
6
7      protected Writer is
8
9          procedure Note_Passing
10             (Cause      : Cause_Of_Termination;
11              Departed   : Task_Id;
12              Event      : Exception_Occurrence);
13             -- Written by someone who's read too much English lit
14
15          procedure Dissemble
16             (Cause      : Cause_Of_Termination;
17              Departed   : Task_Id;
18              Event      : Exception_Occurrence);
19             -- Written by someone who may know more than they're saying
20
21      end Writer;
22
23  end Obituary;
```

Clients can choose among these protected procedures to set a handler for one or more tasks.

The two protected procedures display messages corresponding to the cause of the termination. One procedure prints respectful messages, in the style of someone who's read too much Old English literature. The other prints rather dissembling messages, as if written by someone who knows more than they are willing to say. The point of the difference is that more than one handler can be available to clients, and their choice is made dynamically at run-time.

The package body is structured as follows:

```

with Ada.Text_IO; use Ada.Text_IO;

package body Obituary is

    protected body Writer is
        procedure Note_Passing () is ...
        procedure Dissemble () is ...
    end Writer;

begin -- optional package executable part
    Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
end Obituary;
```

In addition to defining the bodies of the protected procedures, the package body has an executable part. That part is optional, but in this case it is convenient. This executable part calls procedure `Set_Dependents_Fallback_Handler` to apply one of the two handlers.

Because this call happens during library unit elaboration, it sets the fall-back handler for all the tasks in the partition (the program). The effect is global to the partition because library unit elaboration is invoked by the *environment task*, and the environment task is the ultimate master of all application tasks in a partition. Therefore, the fall-back handler is applied to the top of the task dependents hierarchy, and thus to all tasks. The application tasks need not do anything in their source code for the handler to apply to them.

The call to `Set_Dependents_Fallback_Handler` need not occur in this particular package body, or even in a package body at all. But because we want it to apply to all tasks in this specific example, including library tasks, placement in a library package's elaboration achieves that effect.

The observant reader will note the with-clause for `Ada.Text_IO`, included for the sake of references to `Put_Line`. We'll address the ramifications momentarily. Here are the bodies for the two handlers:

Listing 1752: obituary.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Obituary is
4
5     protected body Writer is
6
7         procedure Note_Passing
8             (Cause      : Cause_Of_Termination;
9              Departed   : Task_Id;
10              Event      : Exception_Occurrence)
11         is
12         begin
13             case Cause is
14                 when Normal =>
15                     Put_Line (Image (Departed) &
16                             " went gently into that good night");
17                 when Abnormal =>
18                     Put_Line (Image (Departed) & " was most foully murdered!");
19                 when Unhandled_Exception =>
20                     Put_Line (Image (Departed) &
21                             " was unknot by the much unexpected " &
22                             Exception_Name (Event));
23             end case;
24         end Note_Passing;
25
26         procedure Dissemble
27             (Cause      : Cause_Of_Termination;
28              Departed   : Task_Id;
29              Event      : Exception_Occurrence)
30         is
31         begin
32             case Cause is
33                 when Normal =>
34                     Put_Line (Image (Departed) & " died, naturally.");
35                     Put_Line ("We had nothing to do with it.");
36                 when Abnormal =>
37                     Put_Line (Image (Departed) & " had a tragic accident.");
38                     Put_Line ("We're sorry it had to come to that.");
39                 when Unhandled_Exception =>
40                     Put_Line (Image (Departed) &
41                             " was apparently fatally allergic to " &
42                             Exception_Name (Event));
43             end case;
44         end Dissemble;
```

(continues on next page)

(continued from previous page)

```

45
46     end Writer;
47
48 begin -- optional package executable part
49     Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
50 end Obituary;
```

Now, about those calls to `Ada.Text_IO.Put_Line`. Procedure `Put_Line` is a *potentially blocking* operation. Consequently, a call within a protected operation is a bounded error (see RM 9.5.1(8)) and the resulting execution is not portable. For example, the `Put_Line` calls will likely work as expected on a native OS. However, their execution may do something else on other targets, including raising `Program_Error` if detected. The GNAT bare-metal targets, for example, raise `Program_Error`.

For a portable approach, we move these two blocking calls to a new dedicated task and revise the protected object accordingly. That's portable because a task can make blocking calls.

First, we change `Obituary.Writer` to have a single protected procedure and a new entry. The protected procedure will be used as a termination handler, as before, but does not print the messages. Instead, when invoked by task finalization, the handler enters the parameter values into an internal data structure and then enables the entry barrier on the protected entry. The dedicated task waits on that entry barrier and, when enabled, retrieves the stored values describing a termination. The task can then call `Put_Line` to print the announcement with those values.

Here's the updated `Obituary` package declaration:

Listing 1753: obituary.ads

```

1  with Ada.Exceptions;           use Ada.Exceptions;
2  with Ada.Task_Termination;     use Ada.Task_Termination;
3  with Ada.Task_Identification;  use Ada.Task_Identification;
4  with Ada.Containers.Vectors;
5
6  package Obituary is
7
8      pragma Elaborate_Body;
9
10     Comment_On_Normal_Passing : Boolean := True;
11     -- Do we say anything if the task completed normally?
12
13     type Termination_Event is record
14         Cause      : Cause_Of_Termination;
15         Departed   : Task_Id;
16         Event      : Exception_Id;
17     end record;
18
19     package Termination_Events is new Ada.Containers.Vectors
20         (Positive, Termination_Event);
21
22     protected Writer is
23
24         procedure Note_Passing
25             (Cause      : Cause_Of_Termination;
26              Departed   : Task_Id;
27              Event      : Exception_Occurrence);
28
29         entry Get_Event (Next : out Termination_Event);
30
31     private
```

(continues on next page)

(continued from previous page)

```

32     Stored_Events : Termination_Events.Vector;
33 end Writer;
34
35 end Obituary;

```

As a minor refinement we add the option to not print announcements for normal completions, for those applications that allow task completion.

We must declare the generic container instantiation outside the protected object, an unfortunate limitation of protected objects. We would prefer that clients have no compile-time visibility to it, since it is an implementation artifact.

The updated package body is straightforward:

Listing 1754: obituary.adb

```

1 package body Obituary is
2
3     protected body Writer is
4
5         -----
6         -- Note_Passing --
7         -----
8
9         procedure Note_Passing
10            (Cause      : Cause_Of_Termination;
11             Departed    : Task_Id;
12             Event       : Exception_Occurrence)
13         is
14         begin
15             if Cause = Normal and then
16                 not Comment_On_Normal_Passing
17             then
18                 return;
19             else
20                 Stored_Events.Append
21                     (Termination_Event'(Cause,
22                                         Departed,
23                                         Exception_Identity (Event)));
24             end if;
25         end Note_Passing;
26
27         -----
28         -- Get_Event --
29         -----
30
31         entry Get_Event (Next : out Termination_Event)
32         when
33             not Stored_Events.Is_Empty
34         is
35         begin
36             Next := Stored_Events.First_Element;
37             Stored_Events.Delete_First;
38         end Get_Event;
39
40     end Writer;
41
42 begin -- optional package executable part
43     Set_Dependents_Fallback_Handler (Writer.Note_Passing'Access);
44 end Obituary;

```

In the body of Note_Passing, we store the Exception_Id for the exception occurrence

indicated by Event. That exception occurrence need not be active by the time the task reads the Id for that occurrence.

A new child package declares the task that prints the termination information:

Listing 1755: obituary-output.ads

```

1 package Obituary.Output is
2   pragma Elaborate_Body;
3
4   task Printer;
5
6 end Obituary.Output;
```

In the package body, the task body iteratively suspends on the call to Writer.Get_Event, waiting for a termination handler to make the termination data available. Once it returns from the call, if ever, it simply prints the information and awaits further events:

Listing 1756: obituary-output.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Obituary.Output is
4
5   -----
6   -- Printer --
7   -----
8
9   task body Printer is
10     Next : Termination_Event;
11   begin
12     loop
13       Writer. Get_Event (Next);
14       case Next.Cause is
15         when Normal =>
16           Put_Line (Image (Next.Departed) & " died, naturally.");
17           -- What a difference that comma makes!
18           Put_Line ("We had nothing to do with it.");
19         when Abnormal =>
20           Put_Line (Image (Next.Departed) &
21             " had a terrible accident.");
22           Put_Line ("We're sorry it had to come to that.");
23         when Unhandled_Exception =>
24           Put_Line (Image (Next.Departed) &
25             " reacted badly to " &
26             Exception_Name (Next.Event));
27           Put_Line ("Really, really badly.");
28
29       end case;
30     end loop;
31   end Printer;
32
33 end Obituary.Output;
```

We declared this task in a child package because one can view the Printer and the Writer as parts of a single subsystem, but that structure isn't necessary. An unrelated application task could just as easily retrieve the information stored by the protected Writer object.

Here is a sample demonstration main procedure, a simple test to ensure that termination due to task abort is captured and displayed:

Listing 1757: demo_fallback_handler_abort.adb

```
1 with Obituary.Output; pragma Unreferenced (Obituary.Output);
2 -- otherwise neither package is in the executable
3
4 procedure Demo_Fallback_Handler_Abort is
5
6     task Worker;
7     task body Worker is
8     begin
9         loop -- ensure not already terminated when aborted
10             delay 0.0; -- yield the processor
11         end loop;
12     end Worker;
13
14 begin
15     abort Worker;
16 end Demo_Fallback_Handler_Abort;
```

Note that the nested task would not be accepted under the Ravenscar or Jorvik profiles because those profiles require tasks to be declared at the library level, but that can easily be addressed.

When this demo main is run, the output looks like this:

```
worker_00000174BC68A570 had a terrible accident.
We're sorry it had to come to that.
```

The actual string representing the task identifier will vary with the implementation.

You'll have to use control-c (or whatever is required on your host) to end the program because the Printer task in Obituary.Output runs forever. Many applications run *forever* so that isn't necessarily a problem. That could be addressed if need be.

43.3 Pros

The facility provided by package Ada.Task_Termination allows developers to respond in any way required to task termination. The three causes, normal completion, unhandled exceptions, and task abort are all supported. Significantly, no source code in application tasks is required for the termination support to be applied, other than the isolated calls to set the handlers.

43.4 Cons

On a bare metal target there may be restrictions that limit the usefulness of the facility. For example, on targets that apply the Ravenscar or Jorvik profiles, task abort is not included in the profile and tasks are never supposed to terminate for any reason, including normally. Independent of the profiles, some run-time libraries may not support exception propagation, or even any exception semantics at all.

43.5 Relationship With Other Idioms

None.

43.6 Notes

If you did want to use a generic package to define a task type that is resilient to unhandled exceptions, you could do it like this:

Listing 1758: resilient_workers.ads

```

1 with System;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 generic
5   type Task_Local_State is limited private;
6   with procedure Initialize (This : out Task_Local_State);
7   with procedure Update (This : in out Task_Local_State);
8   with procedure Respond_To_Exception
9     (Current_State : in out Task_Local_State;
10      Error         : Exception_Occurrence);
11 package Resilient_Workers is
12
13   task type Worker
14     (Urgency : System.Priority := System.Default_Priority)
15   with
16     Priority => Urgency;
17
18 end Resilient_Workers;
```

Listing 1759: resilient_workers.adb

```

1 package body Resilient_Workers is
2
3   task body Worker is
4     State : Task_Local_State;
5   begin
6     Recovery : loop
7       begin
8         Initialize (State);
9         Normal : loop
10          Update (State);
11          -- The call above is expected to return, ie
12          -- this loop is meant to iterate
13        end loop Normal;
14      exception
15        when Error : others =>
16          Respond_To_Exception (State, Error);
17      end;
18    end loop Recovery;
19  end Worker;
20
21 end Resilient_Workers;
```

Although this code looks useful, in practice it has issues.

First, in procedure `Initialize`, the formal parameter mode may be a problem. You might need to change the parameter mode from mode `out` to mode `in-out` instead, because recovery from unhandled exceptions will result in another call to `Initialize`. Mode `out` makes sense for the first time `Initialize` is called, but does it make sense for all calls after that? It depends on the application's procedures. The behavior of `Update` may be such that local state should only partially be reset in subsequent calls to `Initialize`.

Furthermore, if `Initialize` must only perform a partial initialization on subsequent calls, the procedure must keep track of the number of calls. That requires a variable declared external to the body of `Initialize`. The additional complexity is unfortunate. We could

perhaps mitigate this problem by having two initialization routines passed to the instantiation: one for full initialization, called only once with mode out for the state, and one for partial initialization, called on each iteration of the Recovery loop with mode in-out for the state:

Listing 1760: resilient_workers.adb

```
1 package body Resilient_Workers is
2
3   task body Worker is
4     State : Task_Local_State;
5   begin
6     Fully_Initialize (State);
7
8     Recovery : loop
9       begin
10         Normal : loop
11           Update (State);
12           -- The call above is expected to return, i.e.
13           -- this loop is meant to iterate
14         end loop Normal;
15       exception
16         when Error : others =>
17           Respond_To_Exception (State, Error);
18       end;
19
20       Partially_Initialize (State);
21     end loop Recovery;
22   end Worker;
23
24 end Resilient_Workers;
```

If both application initialization routines happen to do the same thing, we'd like the developer to be able to pass the same application procedure to both generic formal procedures `Fully_Initialize` and `Partially_Initialize` in the instantiation. But that wouldn't compile because the parameter modes don't match.

Then there's the question of the nature of the task. Is it periodic, or sporadic, or free running? If it is periodic, we need a delay statement in the `Normal` loop to suspend the task for the required period. The generic's task body doesn't do that. The actual procedure passed to `Update` could do the delay, but now, like a single version of `Initialize` required to do both partial and full initialization, it needs additional state declared external to the procedure body (for the `Time` variable used by the absolute delay statement).

Finally, the single generic formal type used to represent the task's local state can be awkward. Having one type for a task's total state is unusual, and aggregating otherwise unrelated types into one isn't good software engineering and doesn't reflect the application domain. Nor is it necessarily trivial to create one type representing a set of distinct variables. For example, some of these stand-alone variables could be objects of indefinite types. Different task objects of a given task type might not agree on those objects' constraints. Furthermore, that awkwardness extends to the procedures that use that single object, in that every procedure except for `Initialize` will likely ignore parts of it.

In summary, the problems are likely more problematic than this generic is worth.

IDIOMS FOR PROTECTED OBJECTS

Note

Parts of this chapter were originally published as the [Blog Post: On the Benefits of Families ... \(Entry Families\)](#)³⁸².

First, a bit of background regarding synchronization in concurrent programming, protected objects, and the `requeue` statement. If you already know this material, feel free to skip ahead to the [Motivation](#) (page 1313) section for this chapter.

Concurrent programming is more complex (and more fun) than purely sequential programming. The cause of this complexity is two-fold: 1) the executing threads' statements are likely interleaved at the assembly language level, and 2) the order of that interleaving is unpredictable. As a result, developers cannot know, in general, where in its sequence of statements any given thread is executing. Developers can only assume that the threads are making finite progress when they execute.

A consequence of unpredictable interleaving is that the bugs specific to this type of programming are timing-dependent. Such bugs are said to be *Heisenbugs* because they "go away when you look at them," i.e., changing the code — adding debugging statements or inserting debugger breakpoints — changes the timing. The bug might then disappear entirely, or simply appear elsewhere in time. We developers must reason about the possible effects of interleaving and design our code to prevent the resulting bugs. (That's why this is fun.) Such bugs are really design errors.

One of these errors is known as a *race condition*. A race condition is possible when multiple threads access some shared resource that requires mutually exclusive access. If we accidentally forget the finite progress assumption we may incorrectly assume that the threads sharing that resource will access it serially. Unpredictable execution interleaving cannot support that assumption.

Race conditions on memory locations are the most common, but the issue is general in nature, including for example hardware devices and OS files. Hence the term "resource."

For example, suppose multiple threads concurrently access an output display device. This device can be ordered to move its cursor to arbitrary points on the display by writing a specific sequence of bytes to it, including the numeric values for X and Y coordinates. A common use is to send the "move cursor to X, Y" sequence and then send the text intended to appear at coordinates X and Y.

Clearly, this device requires each client thread to have mutually exclusive access to the device while sending those two byte sequences. Otherwise, uncoordinated interleaving could result in one thread preempting another thread in the middle of sending those two sequences. The result would be an intermingled sequence of bytes sent to the device. (On a graphics display the chaotic output can be entertaining to observe.)

³⁸² <https://www.adacore.com/blog/on-the-benefits-of-families>

Memory races on variables are less obvious. Imagine two threads, Thread1 and Thread2, that both increment a variable visible to both (an integer, let's say).

Suppose that the shared integer variable has a value of zero. Both threads increment the variable, so after they do so the new value should be two. The compiler will use a hardware register to hold and manipulate the variable's value because of the increased performance over memory accesses. Each thread has an independent copy of the registers, and will perform the same assembly instructions:

1. load a register's value from the memory location containing the variable's value
2. increment the register's value
3. store the register's new value back to the variable's memory location.

The two threads' executions may be interleaved in these three steps. It is therefore possible that Thread1 will execute step 1 and step 2, and then be preempted by the execution of Thread2. Thread2 also executes those two steps. As a result, both threads' registers have the new value of 1. Finally, Thread1 and Thread2 perform the third step, both storing a value of 1 to the variable's memory location. The resulting value of the shared variable will be 1, rather than 2.

Another common design bug is assuming that some required program state has been achieved. For example, for a thread to retrieve some data from a shared buffer, the buffer must not be empty. Some other thread must already have inserted data. Likewise, for a thread to insert some data, the buffer must not be full. Again, the finite progress assumption means that we cannot know whether either of those two states are achieved.

Therefore, interacting threads require two forms of synchronization: *mutual exclusion* and *condition synchronization*. These two kinds of synchronization enable developers to reason rigorously about the execution of their code in the context of the finite progress assumption.

Mutual exclusion synchronization prevents threads that access some shared resource from doing so at the same time, i.e., it provides mutually exclusive access to that resource. The effect is achieved by ensuring that, while any given thread is accessing the resource, that execution will not be interleaved with the execution of any other thread's access to that shared resource.

Condition synchronization suspends a thread until the condition — an arbitrary Boolean expression — is **True**. Only when the expression is (or becomes) **True** can the caller thread be allowed to continue.

A thread-safe bounded buffer is a good example for these two kinds of synchronization. Some threads, the producers, will insert items into the buffer. Other threads, the consumers, will concurrently remove items. The array object representing the buffer contents, as well as the indexes into the array, require mutually exclusive access for both producers and consumers. Furthermore, producers must be blocked (suspended) as long as the given buffer is full, and consumers must be blocked as long as the given buffer is empty.

Concurrent programming languages support mechanisms providing the two forms of synchronization. In some languages these are explicit constructs; other languages take different approaches. In any case, developers can apply these mechanisms to enforce assumptions more specific than simple finite progress.

Ada uses the term **task** rather than thread so we will use that term from here on.

The protected procedures and protected entries declared by a protected object (PO) automatically execute with mutually exclusive access to the entire protected object. No other caller task can be executing these operations at the same time, so execution of the procedure or entry body statements will not be interleaved. (Functions are special because they have read-only access to the data in the PO.) Therefore, there can be no race conditions on the data encapsulated within it. Even if the protected object has no encapsulated data, these operations always execute with mutually exclusive access. During such execution we can say that the PO is locked, speaking informally, because all other caller tasks are held pending.

Protected entries are much like protected procedures, except that entry bodies include a barrier condition that is used to express condition synchronization. The condition is an arbitrary Boolean expression, although there are some restrictions on the content for implementation reasons. Only when the barrier condition is **True** will a caller to the entry be allowed to execute the entry body. Once the body completes, the caller exits and can continue execution outside of the PO. (We'll say more about that later.) For example, entry barriers can express whether a bounded buffer is full or empty, thereby enabling and disabling buffer insertion and removal.

Under some circumstances, an entry may execute a `requeue` statement to reroute the caller to some other entry, for reasons that will be explained shortly, but from the caller task's point of view there is only one call being made.

The `requeue` statement may not be familiar to many readers. To explain its semantics we first need to provide its rationale.

Ada synchronization constructs are based on *avoidance synchronization*, meaning that:

1. the user-written controls that enable/disable the execution of protected entry bodies and task entry accept statements enable them only when they can actually provide the requested service, and
2. that determination is based on information known prior to the execution of the entry body or accept statement.

For example, at runtime, if a bounded buffer is full, that fact can be determined from the buffer's state: is the count of contained items equal to the capacity of the backing array? If so, the user-defined controls disable the operation to insert another value. Likewise, if the buffer is empty, the removal operation is disabled. When we write the buffer implementation we know beforehand what the operations will try to do, so we can write the controls to disallow them at runtime until they can succeed. Most of the time that's sufficient, but not always. If we can't know precisely what the operations will do when we write the code, avoidance synchronization won't suffice.

The **requeue** statement is employed when avoidance synchronization is not sufficient. A task calling an entry that executes a `requeue` statement is much like a person calling a large company on the telephone. Calling the main number connects you to a receptionist (if you're lucky and don't get an annoying menu). If the receptionist can answer your question, they do so and then you both hang up. Otherwise, the receptionist forwards the call to the person they determine that you need to speak with. After doing so, the receptionist hangs up, because from their point of view the call is complete. The call is not complete from your point of view, though, until you finish your conversation with the new person. And of course you may have to wait to speak to that person.

Like the receptionist, the first entry called must take (execute) the call without knowing what the request will be, because the entry barrier cannot reference the entry parameters. The parameter values are only known once the entry body executes. Therefore, the first entry may or may not be able to provide the requested service and allow the caller to return from the call. If not, it requeues the call and finishes, leaving the call still pending on the `requeue` target, i.e., the second entry.

A `requeue` statement is not required in all cases but, as you will see, sometimes it is essential. Note that protected procedures cannot execute `requeue` statements, only protected entries can do so. Protected procedures are appropriate when only mutual exclusion is required (to update encapsulated data).

44.1 Motivation

Of the several highly significant features added to the Ada language over the years, protected objects are one of the most important.

One of the reasons for this prominence is that protected objects make efficient asynchronous task interactions possible. Many, if not most, task interactions are asynchronous, but early Ada had only a synchronous mechanism for communication and synchronization, known informally as the rendezvous. The rendezvous is a high-level, very robust mechanism providing communication and synchronization for two tasks at a time. This mechanism isn't a problem in itself. If the application requires what amounts to an atomic action with two task participants, then the rendezvous meets this requirement nicely.

But, as a synchronous mechanism, the rendezvous is far too expensive when only an asynchronous mechanism (involving only one task) is required. Older mechanisms used for asynchronous interactions, such as semaphores, mutexes, and condition variables, are nowhere near as simple and robust for clients, but are much faster.

In addition, the rendezvous is only available between tasks, meaning that abstractions requiring mutual exclusion and condition synchronization had to be implemented as tasks too. Inserting and removing from a thread-safe buffer, for example, involved expensive task switching between the buffer task, the producer task, and the consumer task. This was the primary source of comparative inefficiency.

There was a non-standard notion of a *passive task* that wasn't actually a thread of control, and therefore did not require task switching, but it was not widely adopted. In that same vein, Ada 80 had a built-in Semaphore task type, intended to be implemented efficiently and used as the name suggests, but mixing the higher-level rendezvous with the much lower-level semaphore abstraction was considered poor language design. It did not survive to the ANSI and first ISO standards. Ultimately, the designers of the first version of Ada thought that processors would become so much faster in the future that the relative inefficiency and semantic mismatch wouldn't matter. Processors did get faster, but the problems still mattered.

Another reason that protected objects are so important is that they are applicable to a wide range of programming domains. Protected objects are critical to concurrent programming, real-time programming, and embedded systems programming with Ada. We've already highlighted their high level, robust support for asynchronous interactions in concurrent programming. For real-time programming, systems of any significant complexity will map deadlines to tasks. Consequently, in such systems the programming model is concurrent programming with the addition of predictability. In these systems protected objects have additional semantics (e.g., priorities) but supply the same benefits as in concurrent programming. For embedded systems programming, protected objects are used to express interrupt handlers, again with added semantics.

Their most important contribution, however, goes beyond direct client use of their automatic mutual exclusion and condition synchronization semantics. Developers can use protected objects to create just about any synchronization and communication protocol imaginable. These include application independent abstractions such as atomic actions, readers-writers locks, mutexes, and so on, but also schemes based on application-specific protocols and data structures. When combined with other language features, such as re-queue, task identifiers, and the overall composition capabilities of the language, the result is a flexible, powerfully expressive facility.

44.2 Implementation

Protected objects are primarily utilized in two ways. We will refer to these two ways as idioms, for the sake of consistency with the rest of this course, although the other idioms in this course are much more narrow in scope. In the first, protected objects encapsulate and manipulate application-specific data. In the second, protected objects are used to create developer-defined synchronization and communication abstractions.

44.2.1 First Idiom Category: Application-Specific Protected Objects

In the first idiom, a given protected object implements all the application-specific functionality for the shared resources it encapsulates. We declare the shared data in the PO private part and declare the protected entries and procedures that manipulate that data in the visible part, as well as functions for reading that data if needed. The compiler won't allow any direct references to the hidden data from outside of the PO; the visible operations must be called by client tasks to manipulate the data. Readers familiar with the classic *monitor* construct will recognize it as the conceptual foundation for protected objects used this way.

For example, let's say we want to protect a product serial number variable from concurrent manipulation by multiple caller tasks. These tasks need to get the next sequential serial number, which entails incrementing the current value each time a task requests the next number. We must prevent the increment from occurring concurrently, otherwise the resulting race condition could occasionally provide incorrect values to the callers. Therefore, the increment will be done inside a protected procedure that provides the current value via parameter and also increments the value before returning. We declare the protected object like so:

```
protected Serial_Number is
  procedure Get_Next (Number : out Positive);
private
  Value : Positive := 1;
end Serial_Number;

protected body Serial_Number is

  procedure Get_Next (Number : out Positive) is
  begin
    Number := Value;
    Value := Value + 1;
  end Get_Next;

end Serial_Number;
```

Whenever any task calls `Serial_Number.Get_Next`, the task will block until it has mutually exclusive access to the PO, and consequently to the `Serial_Number.Value` component. At that point, `Value` is assigned to the formal parameter and then incremented. Once the procedure returns, the caller task can continue execution with their unique serial number copy. No race conditions are possible and the shared serial number value increments safely each time `Get_Next` is called.

Note the robust nature of a protected object's procedural interface: clients simply call the protected procedures, entries, or functions. The called procedure or entry body, when it executes, will always do so with mutually exclusive access. (Functions can have some additional semantics that we can ignore here.) There is no explicit lower level synchronization mechanism for the client to acquire and release. The semantics of protected objects are implemented by the underlying Ada run-time library, hence all error cases are also covered. This procedural interface, with automatic implementation for mutual exclusion, is one of the significant benefits of the monitor construct on which protected objects are based.

44.2.2 Second Idiom Category: Developer-Defined Concurrency Abstractions

In the second idiom, data may be declared in the protected object private part, but they are not application data. Likewise, the protected operations do nothing application specific.

Instead, the PO provides some synchronization (and perhaps communication) protocol that we want to make available to client tasks. These tasks call the protected object's operations

in order to get the protocol's semantics applied to their execution. The data declared in the private part, if any, exist purely for the sake of implementing the intended protocol.

In particular, the protected operations block and release the caller tasks per the new abstraction's semantics. We are using the term *block* loosely here, meaning the caller task is not allowed to return from the call until some condition holds.

These abstractions are frequently declared as protected types rather than anonymously typed protected objects like the `Serial_Number` PO. Protected types are especially preferred when the protocol is application independent and hence reusable. Declaration as a type also provides all the flexibility of types, including the ability to declare as many objects of the type as required and the ability to compose other types using them. Types also allow parameterization via discriminants, if necessary.

The synchronization abstractions may be classic mechanisms long known to the concurrent programming community, for example semaphores, or they may be wholly novel, perhaps based on application-specific contexts and data structures. Very sophisticated abstractions can be expressed, such as atomic actions involving an arbitrary number of tasks. The possibilities are endless.

For example, we could have a protected type that implements the Readers-Writers synchronization protocol. In this protocol only one task at a time can write (update) the state of the shared objects, and writers must wait until there are no readers, but multiple simultaneous readers are allowed as long as there is no writer active. Such a protected object would have multiple protected operations, some to block callers until appropriate for the given read or write action requested, and some to signal the end of the read or write operation so that a pending request (if any) can be granted.

This second idiomatic application of protected objects is extremely useful and therefore common. However, there is also a situation in which we are forced to use it, for the sake of portability. That happens when statements that would otherwise be within a protected operation include a potentially blocking operation. This is a term defined by the language for those constructs that may cause a currently executing caller task to yield the processor to another task. As such, they are not allowed within protected operations, neither directly nor indirectly. To understand why, you need to understand the underlying system approaches available for implementing the mutually exclusive access that protected operations provide automatically.

System Implementation of PO Mutual Exclusion

The underlying run-time library implements the mutual exclusion and thread management semantics for protected objects. Two approaches are known.

One implementation approach, typical when executing on an operating system, uses an explicit locking mechanism provided by the OS. The run-time library code implementing the protected operations first acquires a dedicated OS lock and then later releases it when exiting.

But another approach is available that does not use explicit locks. Instead, mutual exclusion is implemented via priorities, both task priorities and PO priorities. Note that this implementation requires priorities to be defined, execution on a uniprocessor, and the `Ceiling_Locking` policy to have been specified via the `Locking_Policy` pragma.

Specifically, developers assign a priority to each protected object. Each PO priority must be the highest (the *ceiling*) of all the priorities of those tasks that actually call the operations provided by the PO. Consequently, for any given PO, no task that calls that PO will have a higher priority than the PO priority. Because caller tasks inherit the PO priority (immediately), their calls execute with the highest priority of any caller task for that specific PO. Therefore, no other caller task can preempt any current caller executing within the PO. The current caller may be preempted, but not by a task that would also call that same PO. Thus, mutually exclusive access is provided automatically, and very efficiently. This approach has other benefits as well that are not pertinent here.

However, the priority-based implementation cannot work reliably if blocking is allowed within a protected operation. If the current caller could yield the processor inside a protected operation, some other task could then be allowed to continue execution, including possibly a task making a call to that same PO. In that case mutual exclusion would not be provided for that PO.

As a result, the language defines a number of potentially blocking operations and disallows them within protected operations. Any I/O operation is potentially blocking, for example, as are delay statements, but there are others as well. See the Ada RM, section 9.5{34} for the full list.

For example, in the *Dealing with Silent Task Termination idiom* (page 1297) idiom we had an initial implementation of a protected procedure body that called `Ada.Text_IO.Put_Line`:

```

procedure Dissemble
  (Cause      : in Cause_Of_Termination;
   Departed   : in Task_Id;
   Event      : in Exception_Occurrence)
is
begin
  case Cause is
    when Normal =>
      Put_Line (Image (Departed) & " died, naturally.");
      Put_Line ("We had nothing to do with it.");
    when Abnormal =>
      Put_Line (Image (Departed) & " had a tragic accident.");
      Put_Line ("We're sorry it had to come to that.");
    when Unhandled_Exception =>
      Put_Line (Image (Departed) &
        " was apparently fatally allergic to " &
        Exception_Name (Event));
  end case;
end Dissemble;

```

As described in that idiom entry, the above might work, but it is not portable.

As a consequence, we may find ourselves with some statement (e.g., the call to `Put_Line`) that would have been within a protected operation for the sake of mutually exclusive access, but that cannot be included there if the code is to be portable. The statement must be written outside of a PO, not within a protected object's operations, and not in anything called by those protected operations.

Examples for Second Idiom Category

We will use the approach to potentially blocking operations as the first example.

Suppose we are implementing a message logging facility. Any given task executing in the application can write a log message by calling a procedure named `Enter`, defined in package `Log`. The actual messages are values of type **String**:

```

package Log is
  procedure Enter (Log_Entry : String);
end Log;

```

Messages are written to an external file so that they will persist. That file is declared in the package body. Therefore, the package design is an *Abstract Data Machine* (page 1193):

```

with Ada.Text_IO; use Ada.Text_IO;
package body Log is

  Log_File : File_Type;

```

(continues on next page)

(continued from previous page)

```

procedure Enter (Log_Entry : String) is
begin
    Put_Line (Log_File, Msg);
    Flush (Log_File);
end Enter;

begin
    Create (Log_File, Out_File, "log.txt");
end Log;

```

Unfortunately, this won't work reliably in a concurrent program. Multiple tasks may call procedure Enter simultaneously, indirectly making the Log_File object a shared resource. Race conditions are therefore possible when updating the Log_File object via Put_Line. We could employ a protected object to prevent the race condition, but as we saw with protected procedure Disassemble above, placing the call to Put_Line within a protected operation is not portable. We need some other way to ensure mutually exclusive access to the shared file object.

In some programming languages, a *mutex* is used to provide mutually exclusive access (hence the name) to some set of objects that are shared among multiple competing threads. All these threads must follow the same usage pattern:

1. before accessing the shared resource, a thread calls a routine on the mutex in order to block until it is appropriate to continue,
2. upon return from the call that thread executes an arbitrary sequence of statements accessing the resource,
3. after that sequence, the thread calls another operation on the same mutex to signal that some other thread can now be allowed to return from their call in step one.

A mutex must be implemented so that, for any given mutex object, only one caller at a time is allowed to return from the call in step one. Therefore, step one is said to acquire or seize the mutex object, and step three releases it. The result is that only one thread at a time will execute the statements in step two, hence with mutually exclusive access to the manipulated resources.

We can create a protected type providing a basic mutex abstraction. The protected operations will consist of two routines: one to acquire the mutex (step one) and one to release it (step three). Calls to these two PO operations can then bracket an application-specific sequence of statements that manipulate objects requiring mutually exclusive access (step two). But now this bracketed code can include some potentially blocking operations.

```

protected type Mutex is
    entry Acquire;
    procedure Release;
private
    Available : Boolean := True;
end Mutex;

protected body Mutex is

    entry Acquire when Available is
    begin
        Available := False;
    end Acquire;

    procedure Release is
    begin
        Available := True;
    end Release;

```

(continues on next page)

(continued from previous page)

```
end Mutex;
```

Tasks that want to exclusively acquire an object of the Mutex type will call entry Acquire on that PO. Similarly, tasks call protected procedure Release to signal that the logical hold on the PO is no longer required. The component Available is declared within the protected private part, and exists only to implement the mutex semantics.

The gist of the implementation is that acquiring the Mutex object amounts to allowing a caller task to exit their call to the entry Acquire, with any other caller tasks held pending. The entry barrier condition expresses the logic of whether the caller is allowed to continue, via the internal Boolean component Available. There is no actual lock in view here, just the effect of a lock. That effect is achieved via condition synchronization that ensures only one task at a time can return from the Acquire call. All other callers to Acquire are held, suspended, in the entry's queue. When Release is called that Mutex protected object becomes available for locking again.

The following code fragment illustrates using the Mutex type for the sake of controlling access to a shared variable, in this case the file object in the message logging package. Here is the pertinent part of the logging facility's package body:

```
Log_Lock : Mutex;
Log_File : File_Type;

procedure Enter (Log_Entry : String) is
begin
  Log_Lock.Acquire;
  Put_Line (Log_File, Log_Entry);
  Flush (Log_File);
  Log_Lock.Release;
end Enter;
```

The body of procedure Enter first calls Log_Lock.Acquire. The call is not allowed to return until the caller task exclusively holds the logical lock associated with the Log_Lock object. Therefore, every subsequent statement executes with mutual exclusion relative to the Log_Lock object. In this case, there are two such statements, the one that writes the string to the single shared output file and one that flushes any internal buffers associated with the file. They are both potentially blocking operations, but we're not in a protected operation so that's not a problem. Finally, procedure Enter calls Log_Lock.Release to relinquish the current caller task's hold on the Log_Lock mutex. If some other task was waiting to hold the Log_Lock object, that task can now return from its call to Acquire and can execute its update to the log file.

There are issues unaddressed in the three-step client protocol illustrated by the code above, especially error cases. For example, even if an exception is raised in step two, we need to ensure that Release is called with exactly-once semantics. There are other abstractions that address these client usage issues, namely scope-based locking, but we'll ignore them here. See the [Resource Acquisition Is Initialization](#) (page 1279) idiom for the Scope_Lock type.

The implementation of type Mutex above doesn't have quite the full canonical semantics. So far it is really just that of a binary semaphore. In particular, a mutex should only be released by the same task that previously acquired it, i.e., the current owner. We can implement that consistency check in a fuller illustration of this example, one that raises an exception if the caller to Release is not the current owner of the Mutex object.

The new version of type Mutex is declared as follows. The difference is the additional component of type Ada.Task_Identification.Task_Id named Current_Owner. (Assume a use-clause for that package.)

```
protected type Mutex is
  entry Acquire;
  procedure Release;
private
  Available      : Boolean := True;
  Current_Owner : Task_Id := Null_Task_Id;
end Mutex;
```

The updated protected body is as follows:

```
protected body Mutex is

  entry Acquire when Available is
  begin
    Available := False;
    Current_Owner := Acquire'Caller;
  end Acquire;

  procedure Release is
  begin
    if Current_Owner = Current_Task then
      Available := True;
      Current_Owner := Null_Task_Id;
    else
      raise Protocol_Error;
    end if;
  end Release;

end Mutex;
```

Note how entry `Acquire`, when granting the logical lock and releasing the caller, also captures the identity of that caller. Procedure `Release` can verify that identity when it is eventually called, using function `Current_Task` declared in package `Ada.Task_Identification`.

We can build on that version of the type `Mutex` to make a variation named `Reentrant_Mutex`. This type allows a given task to re-acquire a `Reentrant_Mutex` object if that same task is the current owner, i.e., has returned from a previous call to `Acquire` and has not yet called `Release` a matching number of times:

```
protected type Reentrant_Mutex is
  entry Acquire;
  procedure Release;
private

  entry Retry;
  -- Internal target of requeue when the mutex is already owned.

  Depth : Natural := 0;
  -- Number of calls to Seize for a given holder. A value of zero
  -- corresponds no task currently holding the mutex.

  Current_Owner : Task_Id := Null_Task_Id;
  -- The current holder of the mutex, initially none.

end Reentrant_Mutex;
```

We still have the `Current_Owner` component, but we've added a new component to keep track of the *depth* of the current owner's calls. The depth test replaces the simple Boolean test of being available, so the `Available` component is gone. Instead, when the depth is zero the corresponding protected object is available, but it is also available if the current caller of `Acquire` is the current owner from a previous call.

```

protected body Reentrant_Mutex is

  entry Acquire when True is
  begin
    if Current_Owner = Null_Task_Id then
      Current_Owner := Acquire'Caller;
      Depth := 1;
    elsif Current_Owner = Acquire'Caller then
      Depth := Depth + 1;
    else -- held already, but not by current caller
      requeue Retry with abort;
    end if;
  end Acquire;

  procedure Release is
  begin
    if Current_Owner = Current_Task then
      Depth := Integer'Max (0, Depth - 1);
      if Depth = 0 then
        Current_Owner := Null_Task_Id;
      end if;
    else
      raise Protocol_Error;
    end if;
  end Release;

  entry Retry when Depth = 0 is
  begin
    Depth := 1;
    Current_Owner := Retry'Caller;
  end Retry;

end Reentrant_Mutex;

```

The barrier for entry `Acquire` is always set to `True` because the test for availability is not possible until the body begins to execute. If the PO is not available, the caller task is requeued onto the `Retry` entry. (A barrier set to `True` like this is a strong indicator of a requeue operation.) The `Retry` entry will allow a caller to return — from the caller's point of view, a return from the call to `Acquire` — only when no other caller currently owns the PO.

The examples so far exist primarily for providing mutual exclusion to code that includes potentially blocking operations. By no means, however, are these the only examples. Much more sophisticated abstractions are possible.

For example, let's say we want to have a notion of *events* that application tasks can await, suspending until the specified event is *signaled*. At some point, other tasks will signal that these events are ready to be handled by the waiting tasks. Understand that events don't have any state of their own, they either have happened or not, and may happen more than once.

For the sake of discussion let's declare an enumeration type representing four possible events:

```

type Event is (A, B, C, D);

```

These event names are not very meaningful, but they are just placeholders for those that applications would actually define. Perhaps a submersible's code would have events named `Hatch_Open`, `Hatch_Closed`, `Umbilical_Detached`, and so on.

Client tasks can suspend, waiting for an arbitrary event to be signaled, and other tasks can signal the occurrence of events, using a *event manager* that the two sets of tasks reference.

Here's the declaration of the manager type:

```
type Manager is limited private;
```

Type Manager will be fully implemented in the package private part as a protected type.

The type is limited because it doesn't make sense for clients to assign one Manager object to another, nor to compare them via predefined equality. There's another reason you'll see shortly. The type is private because that's the default for good software engineering, and there's no reason to override that default to make the implementation visible to clients. Our API will provide everything clients require, when combined with the capabilities provided by any limited type (e.g., declaring objects, and passing them as parameters).

Tasks can wait for a single event to be signaled, or they can wait for one of several. Similarly, tasks can signal one or more events at a time. Such groups of events are easily represented by an unconstrained array type:

```
type Event_List is array (Positive range <>) of Event;
```

We chose **Positive** as the index subtype because it allows a very large number of components, far more than is likely ever required, and has an intuitive default lower bound of 1. An aggregate value of the array type can then represent multiple events, for example:

```
Event_List'(A, C, D)
```

Given these three types we can define a useful API:

```
procedure Wait
  (This      : in out Manager;
   Any_Of_These : Event_List;
   Enabler    : out Event);

procedure Wait
  (This      : in out Manager;
   This_One : Event);

procedure Signal
  (This      : in out Manager;
   All_Of_These : Event_List);

procedure Signal
  (This      : in out Manager;
   This_One : Event);
```

Here's a task that waits for either event A or B, using a global Controller variable of the Manager type:

```
task body A_or_B_Processor is
  Active : Event;
begin
  loop
    Wait (Controller,
          Any_Of_These => Event_List'(A, B),
          Enabler => Active);
    Put_Line ("A_or_B_Processor responding to event " &
              Active'Image);
  end loop;
end A_or_B_Processor;
```

When the call to Wait returns, at least one of either A or B has been signaled. One of those signaled events was selected and returned in the Enabler parameter. That selected event is no longer signaled when the call returns and will stay that way until another call

to procedure `Signal` changes it. The other event in the list is not affected, whether or not it was also signaled.

A signaling task could use the API to signal one event:

```
Signal (Controller, This_One => B);
```

or just:

```
Signal (Controller, B);
```

To signal multiple events:

```
Signal (Controller, All_Of_These => Event_List'(A, C, D));
```

Now let's consider the Manager implementation. As this is a concurrent program, we need it to be thread-safe. We've declared the Manager type as limited, so either a task type or a protected type would be allowed as the type's completion. (That's the other reason the type is limited.) There's no need for this manager to do anything active, it just suspends some tasks and resumes others when invoked. Therefore, a protected type will suffice, rather than a task's active thread of control.

Clearly, tasks that await events must block until a requested event has been signaled, assuming it was not already signaled when the call occurred, so a protected procedure won't suffice. Protected procedures only provide mutual exclusion, whereas protected entries can suspend a caller on a condition. Therefore, we'll use a protected entry for the waiters to call. As you will see later, there is another reason to use protected entries here.

Inside the Manager protected type we need a way to represent whether events have been signaled. We can use an array of Boolean components for this purpose, with the events as the indexes. For any given event index value, if the corresponding array component is **True** that event has been signaled, otherwise it has not.

```
type Event_States is array (Event) of Boolean;
Signaled : Event_States := (others => False);
```

Thus, for example, if `Signaled (B)` is **True**, a task that calls `Wait` for B will be able to return. Otherwise, that task will be blocked and cannot return from the call. Later another task will set `Signaled (B)` to **True**, and then the waiting task can be unblocked.

Since an aggregate can also contain only one component if desired, we can use a single set of protected routines for waiting and signaling in the Manager protected type. We don't need one set of routines for waiting and signaling a single event, and another set of routines for waiting and signaling multiple events. Here then is the visible part:

```
protected type Manager is
  entry Wait
    (Any_Of_These : Event_List;
     Enabler      : out Event);

  procedure Signal (All_Of_These : Event_List);
private
  ...
end Manager;
```

Both the entry and the procedure take an argument of the array type, indicating one or more client events. The entry, called by waiting tasks, also has an output argument, `Enabler`, indicating which specific event enabled the task to resume, i.e., which event was found signaled and was selected to unblock the task. We need that parameter because the task

may have specified that any one of several events would suffice, and more than one could have been signaled.

The bodies of our API routines are then just calls into the protected Manager that is passed as the first argument. For example, here are two of the four:

```
procedure Wait
  (This      : in out Manager;
   Any_Of_These : Event_List;
   Enabler    : out Event)
is
begin
  This.Wait (Any_Of_These, Enabler);
end Wait;

procedure Signal
  (This      : in out Manager;
   This_One : Event)
is
begin
  This.Signal (Event_List'(1 => This_One));
end Signal;
```

Now let's examine the implementation of the protected type. The visible part is repeated here:

```
protected type Manager is

  entry Wait
    (Any_Of_These : Event_List;
     Enabler      : out Event);

  procedure Signal (All_Of_These : Event_List);

private
  ...
end Manager;
```

The entry `Wait` suspends callers until one of the requested events is signaled, as specified by the argument `Any_Of_These`. Normally we'd expect to use the entry barrier to express this behavior by querying the events' state array. If one of the requested events is **True** the barrier would allow the call to execute and complete. However, barriers do not have compile-time visibility to the entry parameters, so the parameters cannot be referenced in the barriers. This situation calls for a `requeue` statement.

Because `Wait` always takes a call, the entry barrier is just hard-coded to **True**. (As mentioned earlier, that's always a strong indication that `requeue` is involved.) Even though this barrier always allows a call, much like a protected procedure, we must use an entry because only protected entries can `requeue` callers.

Inside the entry body the specified events' states are checked, looking for one that is **True**. If one is found, the entry body completes and the caller returns to continue further, responding to the found event. If no requested event is **True**, though, we cannot let the caller continue. We block it by `requeueing` the caller on to another entry. Eventually that other entry will allow the caller to return, when an awaited event finally becomes **True** via `Signal`.

Here then is the full declaration for the protected type `Manager`, including the array type declaration that cannot be internal to the protected type:

```
type Event_States is array (Event) of Boolean;
```

(continues on next page)

(continued from previous page)

```

protected type Manager is

    entry Wait
        (Any_Of_These : Event_List;
         Enabler       : out Event);

    procedure Signal (All_Of_These : Event_List);

private

    Signaled          : Event_States := (others => False);
    Prior_Retry_Calls : Natural := 0;

    entry Retry
        (Any_Of_These : Event_List;
         Enabler       : out Event);

end Manager;

```

The private part contains the event states component, a management component, and the other entry, `Retry`, onto which we will requeue when necessary. Note that this other entry is only meant to be called by a requeue from the visible entry `Wait`, so we declare it in the private part to ensure there are no other calls to it. That informs the reader, but also the maintainer, who in the future might be tempted to call it in some other context.

The routine that checks for an existing signaled event is internal to the protected type and is declared as follows:

```

procedure Check_Signaled
    (These      : Event_List;
     Enabler    : out Event;
     Found      : out Boolean);

```

The procedure examines the events specified in the formal parameter `These` to see if any of them are currently signaled, i.e., have a value of **True**. If it finds one, `Enabler` is set to that event value and `Found` is set to **True**. Otherwise, `Found` is set to **False** and `Enabler` is set to the value `Event'First`. The value assigned to `Enabler` in that case is arbitrary, but the assignment itself is important. Assigning a value prevents the actual parameter from becoming undefined upon return. `Enabler` will only be evaluated when `Found` returns **True** so the arbitrary value will be ignored.

Here's the body of the entry `Wait`, containing a call to `Check_Signaled` and the requeue statement. Note that the formal parameter `Wait.Enabler` is passed as the actual parameter to `Check_Signaled.Enabler`.

```

entry Wait
    (Any_Of_These : Event_List;
     Enabler      : out Event)
when
    True
is
    Found_Awaited_Event : Boolean;
begin
    Check_Signaled (Any_Of_These, Enabler, Found_Awaited_Event);
    if not Found_Awaited_Event then
        requeue Retry;
    end if;
end Wait;

```

The hard-coded entry barrier (**when True**) always allows a caller to execute, subject to the mutual exclusion requirement. If `Check_Signaled` doesn't find one of the specified events

signaled, we requeue the caller to the Retry entry. (The Wait entry parameters go to the Retry entry, transparently.) On the other hand, if Check_Signaled did find a specified event signaled, we just exit the entry, the formal parameter Enabler having been set already by the call to the internal procedure.

Eventually, presumably, an awaited **False** event will become **True**. That happens when Signal is called:

```
procedure Signal (All_Of_These : Event_List) is
begin
  for C of All_Of_These loop
    Signaled (C) := True;
  end loop;
  Prior_Retry_Calls := Retry'Count;
end Signal;
```

After setting the specified events' states to **True**, Signal captures the number of queued callers waiting on Retry. (The component Prior_Retry_Calls is an internal component declared in the protected type. The value is never presented to callers, but is, instead, used only to manage callers.)

At long last, here's the body of Retry:

```
entry Retry
  (Any_Of_These : Event_List;
   Enabler      : out Event)
when
  Prior_Retry_Calls > 0
is
  Found_Signaled_Event : Boolean;
begin
  Prior_Retry_Calls := Prior_Retry_Calls - 1;
  Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
  if not Found_Signaled_Event then
    requeue Retry;
  end if;
end Retry;
```

When a protected procedure or entry completes their sequence of statements, the run-time system re-evaluates all the object's entry barriers, looking for an open (**True**) barrier with a caller queued, waiting. If one is found, that entry body is allowed to execute on behalf of that caller. On exit, the evaluation / execution process repeats. This process is known as a protected action and is one reason protected objects are so expressive and powerful. The protected action continues iterating, executing enabled entry bodies on behalf of queued callers, until either no barriers are open or no open barriers have callers waiting. Note that one of these entries may enable the barrier condition of some other entry in that same PO.

Therefore, when procedure Signal sets Prior_Retry_Calls to a value greater than zero and then completes, the protected action allows Retry to execute. Furthermore, Retry continues to execute, attempting to service all the prior callers in the protected action, because its barrier becomes **False** only when all those prior callers have been serviced.

For each caller, Retry attempts the same thing Wait did: if a requested event is **True** the caller is allowed to return from the call. Otherwise, the caller is requeued onto Retry. So yes, Retry requeues the caller onto itself! Doing so is not inherently a problem, but in this particular case a caller would continue to be requeued indefinitely when the requested event is **False**, unless something prevents that from happening. That's the purpose of the count of prior callers. Only that number of callers are executed by the body of Retry in the protected action. After that the barrier is closed by Prior_Retry_Calls becoming zero, the protected action ceases when the entry body exits, and any unsatisfied callers remain queued.

All well and good, but have you noticed the underlying assumption? The code assumes

that unsatisfied callers are placed onto the entry queue at the end of the queue, i.e., in FIFO order. Consequently, they are not included in the value of the `Prior_Retry_Calls` count and so do not get executed again until `Signal` is called again. But suppose we have requested elsewhere that entry queues (among other things) are ordered by caller priority? We'd want that for a real-time system. But then a requeued caller would not go to the back of the entry queue and could, instead, execute all over again, repeatedly, until the prior caller count closed the entry.

If priority queuing might be used, we must change the internal implementation so that the queuing policy is irrelevant. We'll still have `Wait` do a requeue when necessary, but no requeue will ever go to the same entry executing the requeue statement. Therefore, the entry queuing order won't make a difference. An entry family facilitates that change, and rather elegantly, too.

An entry family is much like an array of entries, each one identical to the others. To work with any one of the entries we specify an index, as with an array. For example, here's a requeue to `Retry` as a member of an entry family, with `Active_Retry` as the index:

```
requeue Retry (Active_Retry)
```

In the above, the caller uses the value of `Active_Retry` as an index to select a specific entry in the family.

The resulting changes to the `Manager` type are as follows:

```
type Retry_Entry_Id is mod 2; -- hence 0 .. 1
type Retry_Barriers is array (Retry_Entry_Id) of Boolean;

protected type Manager is
  ... as before
private

  Signaled      : Event_States := (others => False);
  Retry_Enabled : Retry_Barriers := (others => False);
  Active_Retry  : Retry_Entry_Id := Retry_Entry_Id'First;

  entry Retry (Retry_Entry_Id)
    (Any_Of_These : Event_List;
     Enabler      : out Event);

end Manager;
```

Our entry family index type is `Retry_Entry_Id`. We happen to need two entries in this implementation, so a modular type with two values will suffice. Modular arithmetic will also express the logic nicely, as you'll see. The component `Active_Retry` is of this type, initialized to zero.

The entry `Retry` is now a family, as indicated by the entry declaration syntax specifying the index type `Retry_Entry_Id` within parentheses. Each entry has the same parameters as any others in the family, in this case the same parameters as in the previous implementation.

We thus have two `Retry` entries so that, at any given time, one of the entries can requeue onto the other one, instead of onto itself. An entry family makes that simple to express.

At runtime, one of the `Retry` entries will field requeue calls from `Wait` and will be the entry enabled by `Signal`. That entry is designated the *active* retry target, via the index held in the component `Active_Retry`.

Here's the updated body of `Wait`:

```
entry Wait
  (Any_Of_These : Event_List;
```

(continues on next page)

(continued from previous page)

```

    Enabler      : out Event)
when
    True
is
    Found_Signaled_Event : Boolean;
begin
    Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
    if not Found_Signaled_Event then
        requeue Retry (Active_Retry) with abort;
    end if;
end Wait;

```

The body is as before, except that the requeue target depends on the value of `Active_Retry`. (We'll discuss `with abort` later.)

When `Signal` executes, it now enables the *active retry* entry barrier:

```

procedure Signal (All_Of_These : Event_List) is
begin
    for C of All_Of_These loop
        Signaled (C) := True;
    end loop;
    Retry_Enabled (Active_Retry) := True;
end Signal;

```

The barrier component `Retry_Enabled` is now an array, using the same index type as the entry family.

The really interesting part of the implementation is the body of `Retry`, showing the expressive power of the language. The entry family member enabled by `Signal` goes through all its pending callers, attempting to satisfy them and requeuing those that it cannot. But instead of requeuing onto itself, it requeues them onto the other entry in the family. As a result, the ordering of the queues is immaterial. Again, the entry family makes this easy to express:

```

entry Retry (for K in Retry_Entry_Id)
    (Any_Of_These : Event_List;
     Enabler      : out Event)
when
    Retry_Enabled (K)
is
    Found_Signaled_Event : Boolean;
begin
    Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
    if Found_Signaled_Event then
        return;
    end if;
    -- otherwise...
    if Retry (K).Count = 0 then -- current caller is last one present
        -- switch to the other Retry family member for
        -- subsequent retries
        Retry_Enabled (K) := False;
        Active_Retry := Active_Retry + 1;
    end if;
    -- NB: K + 1 wraps around to the other family member
    requeue Retry (K + 1) with abort;
end Retry;

```

Note the first line:

```

entry Retry (for K in Retry_Entry_Id)

```

as well as the entry barrier (before the reserved word **is**):

```
when Retry_Enabled (K)
```

K is the entry family index, in this case iterating over all the values of `Retry_Entry_Id` (i.e., 0 .. 1).

We don't have to write a loop checking each family member's barrier; that happens automatically, via K. When a barrier at index K is found to be **True**, that corresponding entry can execute a prior caller.

Note the last statement, the one performing the requeue:

```
requeue Retry (K + 1) with abort;
```

Like the `Active_Retry` component, the index K is of the modular type with two possible values, so `K + 1` is always the *other* entry of the two. The addition wraps around, conveniently. As a result, the requeue is always onto the other entry, never itself, so the entry queue ordering makes no difference.

The **with abort** syntax can be read as "with abort enabled for the requeued caller task." Ordinarily, an aborted task that is suspended on an entry queue is removed from that queue. That removal is allowable in this version of protected type Manager, unlike the earlier FIFO version, because we are not using the count of prior callers to control the number of iterations in the protected action involving `Retry`. In the FIFO implementation we could not allow requeued callers to be removed from the `Retry` queue because the count of prior callers would no longer match the number of queued callers actually present. The protected action would await a caller that would never execute. In this more robust implementation that cannot happen, so it is safe to allow aborted tasks to be removed from the `Retry` queue.

Note that we do still check the count of pending queued callers, we just don't capture it and use it to control the number of iterations in the protected action. If we've processed the last caller for member K, we close member K's barrier immediately and then set the active member index to the other entry member. Consequently, a subsequent call to `Wait` will requeue to the other entry family member and `Signal` will, eventually, enable it.

Because we did not make the implementation visible to the package's clients, our internal changes will not require users to change any of their code.

Note that both the Ravenscar and Jorvik profiles allow entry families, but Ravenscar allows only one member per family because only one entry is allowed per protected object. Such an entry family doesn't provide any benefit over a single entry declaration. Jorvik allows multiple entry family members because it allows multiple entries per protected object. However, neither profile allows requeue statements, for the sake of simplifying the underlying run-time library implementation.

The full version using the entry-family approach is provided at the end of this text. Note that we have used a generic package so that we can factor out the specific kind of events involved, via the generic formal type. As long as the generic actual type is a discrete type the compiler will be happy. That correspondence is essential because we use the event type as an index for the array type `Event_States`.

```
generic
  type Event is (<>);
package Event_Management is

  type Manager is limited private;

  ...

private
```

(continues on next page)

(continued from previous page)

```

    type Event_States is array (Event) of Boolean;
    ...
end Event_Management;

```

Here is a small demonstration program. As before, we just have some simple event names to await and signal. We instantiate the generic package `Event_Management` with that `Event` type, and also the generic package `Ada.Numerics.Discrete_Random` so that we can randomly generate events to test the `Event_Management` instance.

```

-- Make the protected entry queues not be FIFO ordered, to
-- demonstrate that the type Manager handles this case too.
pragma Queuing_Policy (Priority_Queueing);

with Ada.Text_IO;    use Ada.Text_IO;
with Event_Management;

with Ada.Numerics.Discrete_Random;

procedure Demo_Events is

    type Event is (A, B, C, D);

    package Events is new Event_Management (Event);
    use Events;

    package Arbitrary_Event is
        new Ada.Numerics.Discrete_Random (Event);
    use Arbitrary_Event;

    G : Arbitrary_Event.Generator;

    Controller : Events.Manager;

    -- Tasks to await the events being signaled.
    --
    -- We give them priorities to exercise the priority-based
    -- implementation, but the values are arbitrary.

    task A_or_B_Processor with Priority => 5;
    task C_Processor      with Priority => 6;
    task D_Processor      with Priority => 7;

    -----
    -- A_or_B_Processor --
    -----

    task body A_or_B_Processor is
        Active : Event;
    begin
        loop
            Wait (Controller,
                  Any_Of_These => Event_List'(A, B),
                  Enabler => Active);
            Put_Line ("A_or_B_Processor responding to event " &
                      Active'Image);
        end loop;
    end A_or_B_Processor;

    -----
    -- C_Processor --

```

(continues on next page)

(continued from previous page)

```

-----

task body C_Processor is
begin
  loop
    Wait (Controller, C);
    Put_Line ("C_Processor responding to event C");
  end loop;
end C_Processor;

-----
-- D_Processor --
-----

task body D_Processor is
begin
  loop
    Wait (Controller, D);
    Put_Line ("D_Processor responding to event D");
  end loop;
end D_Processor;

begin
  loop
    Signal (Controller, Random (G));
    -- The tasks have priority for the sake of realism
    -- since the queues are now ordered by priority.
    -- However, we don't want any one task to
    -- monopolize the output, so for the sake of the
    -- demonstration we give the other tasks time to
    -- suspend on their calls to Wait too. The delay
    -- also makes the output easier to read.
    delay 0.5;
  end loop;
end Demo_Events;

```

When executed, each task iteratively prints a message indicated that it is responding to one of the awaited events. One of the tasks waits for one of two specified events, and the other two tasks wait for a single event each. The main procedure signals events at random. The demo runs forever so you'll have to kill it manually.

Each task writes to `Standard_Output`. Strictly speaking, this tasking structure allows race conditions on that shared (logical) file, but this is just a simple demo of the event facility so it is not worth bothering to prevent them. For the same reason, we didn't declare a task type parameterized with a discriminant for those tasks that await a single event.

44.2.3 Concurrent Programming

Concurrent programming applications will likely use both idioms. Thread-safe buffers are extremely common, for example. But in addition, potentially blocking operations are sometimes necessary within regions of code that require mutually exclusive access.

44.2.4 Real-Time Programming

As we mentioned in the introduction, protected objects provide the same benefits for real-time programming as they provide for concurrent programming, albeit with additional semantics. Those additions include execution with a priority, in particular. Clients will assign a ceiling priority to each protected object, as described in the *System Implementation of PO Mutual Exclusion* (page 1316) section above. The purpose is to limit the blocking experienced by tasks, along with other task interaction benefits on a uniprocessor.

The GNAT package hierarchy includes a thread-safe bounded buffer abstraction that can be used in real-time applications. The protected type is declared within a generic package, like so:

```
protected type Bounded_Buffer
  (Capacity : Positive;
   Ceiling  : System.Priority)
with
  Priority => Ceiling
is
  ...
private
  ...
end Bounded_Buffer;
```

The two discriminants allow the type to be parameterized when clients declare objects of the type. In this case, the Capacity discriminant will be given a value specifying the maximum number of Element values that the object should be able to contain. More to the point here, the Ceiling discriminant specifies the priority to be given to the protected object itself.

44.2.5 Embedded Systems Programming

In the canonical model for handling interrupts in Ada, the handlers are protected procedures. The enclosing PO is again given a priority, as for real-time programming, but now the priority level is that of the Interrupt_Priority subtype. The handlers are invoked by the hardware via the run-time library and execute at the priority specified. This is essentially use of the first idiom for protected objects, even though the encapsulated application data is hardware oriented.

That's the canonical model, and hence the portable approach, but other approaches are possible. For example, if the target and OS allow it, a developer can set up the system to directly vector to a non-protected procedure. However, doing so is not portable, loses the benefits of the integration with the priority scheme, and almost certainly includes limitations on the operations allowed within the procedure body.

For further discussion, see the [Interrupt Handling](#) (page 1261) idiom entry.

44.3 Pros

Protected objects extend prior research in concurrent programming language constructs, specifically the monitor construct that replaces semaphores, mutexes, and condition variables. Condition synchronization is simply stated as a Boolean expression on an entry, with caller suspension and resumption handled automatically. The monitor's explicit mechanism for local caller suspension and resumption is no longer required. Furthermore, protected action semantics make the code simpler than the combination of condition variables with mutexes, including the need for the looping idiom in the PThreads model, because when the condition expressed by the barrier becomes **True** the awakened caller is guaranteed to hold the mutual exclusion lock.

Protected objects add asynchronous communication and synchronization to the existing synchronous mechanism of Ada 83, addressing a serious deficiency for both concurrent and real-time programming in Ada. They are also critical to embedded systems programming with Ada.

Most importantly, developers can use protected objects (types) to create just about any synchronization and communication protocol imaginable. Especially when combined with other language features, the result is a flexible, extremely expressive facility.

44.4 Cons

The private part of a protected definition can contain only declarations for protected operations and component declarations. This limitation leads to declarations, such as the array type `Event_States`, that are purely implementation artifacts but cannot be hidden inside the private part. These artifacts will usually be declared immediately before the protected object or type, thus making them compile-time visible to clients whenever the protected type or object is visible to clients. Note that anonymously-typed array objects are not allowed in the private part. You will understand why these limitations exist when you consider that protected objects, when first conceived, were known as *protected records*. They have only slightly more declarative options than those of record types.

Protected objects have more capabilities than semaphores, mutexes, and condition variables. As a consequence, they may have more run-time overhead, but not much. For the automatic mutual exclusion implementation, the expense can be literally zero when priorities are used instead of actual locks for the implementation.

44.5 Relationship With Other Idioms

None

44.6 Notes

1. Thanks to Andrei Gritsenko (Андрей Гриценко@disqus_VErl9jPNvR) for suggesting a nice simplification of the FIFO version of the event waiting facility.
2. For more on tasking and topics like this, including examples of the second idiom, see the book by Burns and Wellings, *Concurrent and Real-Time Programming In Ada*, Cambridge University Press, 2007. Yes, 2007, but it is excellent and remains directly applicable today. The implementation of the event manager is based on their `Resource_Controller` example, for example.

44.7 Full Code for Event_Management Generic Package

The full implementation of the approach that works regardless of whether the queues are FIFO ordered is provided below. Note that it includes some defensive code that we did not mention above, for the sake of simplicity. See in particular procedures `Signal` and `Wait` that take an `Event_List` as inputs.

When compiling this generic package, you may get warnings indicating that the use of parentheses for aggregates is an obsolete feature and that square brackets should be used instead. Feel free to ignore them. Parentheses are not obsolete, neither in a practical sense nor in the language standard's sense of being obsolescent. There are indeed cases where square brackets are better, or even required, but those situations don't appear here.

Listing 1761: event_management.ads

```

1  -- This package provides a means for blocking a calling task
2  -- until/unless any one of an arbitrary set of "events" is
3  -- "signaled."
4
5  -- NOTE: this implementation allows either priority-ordered or
6  -- FIFO-ordered queuing.
7
8  generic
9  type Event is (<>);
```

(continues on next page)

(continued from previous page)

```

10 package Event_Management is
11
12     type Manager is limited private;
13
14     type Event_List is array (Positive range <>) of Event;
15
16     procedure Wait
17         (This      : in out Manager;
18          Any_Of_These : Event_List;
19          Enabler    : out Event)
20     with
21         Pre => Any_Of_These'Length > 0;
22     -- Block until/unless any one of the events in Any_Of_These has
23     -- been signaled. The one enabling event will be returned in the
24     -- Enabler parameter, and is cleared internally as Wait exits.
25     -- Any other signaled events remain signaled. Note that,
26     -- when Signal is called, the events within the aggregate
27     -- Any_of_These are checked (for whether they are signaled)
28     -- in the order they appear in the aggregate. We use a precondition
29     -- on Wait because the formal parameter Enabler is mode out, and type
30     -- Event is a discrete type. As such, if there was nothing in the list
31     -- to await, the call would return immediately, leaving Enabler's value
32     -- undefined.
33
34     procedure Wait
35         (This      : in out Manager;
36          This_One   : Event);
37     -- Block until/unless the specified event has been signaled.
38     -- This procedure is a convenience routine that can be used
39     -- instead of an aggregate with only one event component.
40
41     procedure Signal
42         (This      : in out Manager;
43          All_Of_These : Event_List);
44     -- Indicate that all of the events in All_Of_These are now
45     -- signaled. The events remain signaled until cleared by Wait.
46     -- We don't use a similar precondition like that of procedure
47     -- Wait because, for Signal, doing nothing is what the empty
48     -- list requests.
49
50     procedure Signal
51         (This      : in out Manager;
52          This_One   : Event);
53     -- Indicate that event This_One is now signaled. The event
54     -- remains signaled until cleared by Wait. This procedure is a
55     -- convenience routine that can be used instead of an aggregate
56     -- with only one event component.
57
58 private
59
60     type Event_States is array (Event) of Boolean;
61
62     type Retry_Entry_Id is mod 2;
63
64     type Retry_Barriers is array (Retry_Entry_Id) of Boolean;
65
66     protected type Manager is
67         entry Wait
68             (Any_Of_These : Event_List;
69              Enabler      : out Event);
69         procedure Signal (All_Of_These : Event_List);
70

```

(continues on next page)

(continued from previous page)

```

71 private
72     Signaled      : Event_States := (others => False);
73     Retry_Enabled : Retry_Barriers := (others => False);
74     Active_Retry  : Retry_Entry_Id := Retry_Entry_Id'First;
75     entry Retry (Retry_Entry_Id)
76         (Any_Of_These : Event_List;
77          Enabler       : out Event);
78 end Manager;
79
80 end Event_Management;

```

And the generic package body:

Listing 1762: event_management.adb

```

1  package body Event_Management is
2
3      -----
4      -- Wait --
5      -----
6
7      procedure Wait
8          (This      : in out Manager;
9           Any_Of_These : Event_List;
10          Enabler    : out Event)
11      is
12      begin
13          This.Wait (Any_Of_These, Enabler);
14      end Wait;
15
16      -----
17      -- Wait --
18      -----
19
20      procedure Wait
21          (This      : in out Manager;
22           This_One  : Event)
23      is
24          Unused : Event;
25      begin
26          This.Wait (Event_List'(1 => This_One), Unused);
27      end Wait;
28
29      -----
30      -- Signal --
31      -----
32
33      procedure Signal
34          (This      : in out Manager;
35           All_Of_These : Event_List)
36      is
37      begin
38          -- Calling Manager.Signal has an effect even when the list
39          -- is empty, albeit minor, so we don't call it in that case
40          if All_Of_These'Length > 0 then
41              This.Signal (All_Of_These);
42          end if;
43      end Signal;
44
45      -----
46      -- Signal --

```

(continues on next page)

(continued from previous page)

```

47  -----
48
49  procedure Signal
50      (This      : in out Manager;
51       This_One  : Event)
52  is
53  begin
54      This.Signal (Event_List'(1 => This_One));
55  end Signal;
56
57  -----
58  -- Manager --
59  -----
60
61  protected body Manager is
62
63      procedure Check_Signaled
64          (These      : Event_List;
65           Enabler    : out Event;
66           Found      : out Boolean);
67
68      -----
69      -- Wait --
70      -----
71
72      entry Wait
73          (Any_Of_These : Event_List;
74           Enabler      : out Event)
75      when
76          True
77      is
78          Found_Signaled_Event : Boolean;
79      begin
80          Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
81          if not Found_Signaled_Event then
82              requeue Retry (Active_Retry) with abort;
83          end if;
84      end Wait;
85
86      -----
87      -- Signal --
88      -----
89
90      procedure Signal (All_Of_These : Event_List) is
91      begin
92          for C of All_Of_These loop
93              Signaled (C) := True;
94          end loop;
95          Retry_Enabled (Active_Retry) := True;
96      end Signal;
97
98      -----
99      -- Retry --
100      -----
101
102      entry Retry (for K in Retry_Entry_Id)
103          (Any_Of_These : Event_List;
104           Enabler      : out Event)
105      when
106          Retry_Enabled (K)
107      is

```

(continues on next page)

(continued from previous page)

```

108     Found_Signaled_Event : Boolean;
109 begin
110     Check_Signaled (Any_Of_These, Enabler, Found_Signaled_Event);
111     if Found_Signaled_Event then
112         return;
113     end if;
114     -- otherwise...
115     if Retry (K)'Count = 0 then -- current caller is last one
116         -- switch to the other Retry family member for
117         -- subsequent retries
118         Retry_Enabled (K) := False;
119         Active_Retry := Active_Retry + 1;
120     end if;
121     -- NB: K + 1 wraps around to the other family member
122     requeue Retry (K + 1) with abort;
123 end Retry;
124
125 -----
126 -- Check_Signaled --
127 -----
128
129 procedure Check_Signaled
130   (These      : Event_List;
131    Enabler    : out Event;
132    Found      : out Boolean)
133 is
134 begin
135     for C of These loop
136         if Signaled (C) then
137             Signaled (C) := False;
138             Enabler := C;
139             Found := True;
140             return;
141         end if;
142     end loop;
143     Enabler := Event'First; -- arbitrary, avoids undefined value
144     Found := False;
145 end Check_Signaled;
146
147 end Manager;
148
149 end Event_Management;

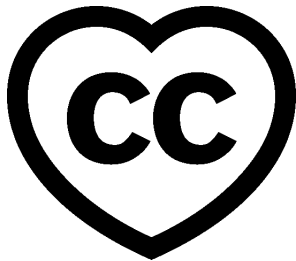
```


Part IV

Ada for the C++ or Java Developer

Copyright © 2013 – 2023, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)³⁸³



This document will present the Ada language using terminology and examples that are familiar to developers that understand the C++ or Java languages.

This document was prepared by Quentin Ochem, with contributions and review from Richard Kenner, Albert Lee, and Ben Brosgol.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](https://learn.adacore.com/zip/learning-ada_code.zip)³⁸⁴. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

³⁸³ <http://creativecommons.org/licenses/by-sa/4.0>

³⁸⁴ https://learn.adacore.com/zip/learning-ada_code.zip

PREFACE

Nowadays it seems like talking about programming languages is a bit *passé*. The technical wars of the past decade have subsided and today we see a variety of high-level and well-established languages offering functionality that can meet the needs of any programmer.

Python, Java, C++, C#, and Visual Basic are recent examples. Indeed, these languages make it easier to write code very quickly, are very flexible, offer features with highly dynamic behavior, and some even allow compilers to deduce the developer's probable intent.

Why, then, talk about yet another language? Well, by addressing the general programming market, the aforementioned languages have become poorly suited for working within the domain of high-integrity systems. In highly reliable, secure and safe applications such as those found in and around airplanes, rockets, satellites, trains, and in any device whose failure could jeopardize human life or critical assets, the programming languages used must support the high standard of software engineering necessary to maintain the integrity of the system.

The concept of verification — the practice of showing that the system behaves and performs as intended — is key in such environments. Verification can be accomplished by some combination of review, testing, static analysis, and formal proof techniques. The increasing reliance on software and increasing complexity of today's systems has made this task more difficult. Technologies and practices that might have been perfectly acceptable ten or fifteen years ago are insufficient today. Thankfully, the state of the art in analysis and proof tools and techniques has also advanced.

The latest revisions of the Ada language, Ada 2005 and Ada 2012, make enhanced software integrity possible. From its inception in the 1980s, Ada was designed to meet the requirements of high-integrity systems, and continues to be well-suited for the implementation of critical embedded or native applications. And it has been receiving increased attention recently. Every language revision has enhanced expressiveness in many areas. Ada 2012, in particular, has introduced new features for contract-based programming that are valuable to any project where verification is part of the engineering lifecycle. Along with these language enhancements, Ada compiler and tool technology has also kept pace with general computing developments over the past few years. Ada development environments are available on a wide range of platforms and are being used for the most demanding applications.

It is no secret that we at AdaCore are very enthusiastic about Ada, but we will not claim that Ada is always the solution; Ada is no more a silver bullet than any other language. In some domains other languages make sense because of the availability of particular libraries or development frameworks. For example, C++ and Java are considered good choices for desktop programs or applications where a shortened time to market is a major objective. Other areas, such as website programming or system administration, tend to rely on different formalisms such as scripting and interpreted languages. The key is to select the proper technical approach, in terms of the language and tools, to meet the requirements. Ada's strength is in areas where reliability is paramount.

Learning a new language shouldn't be complicated. Programming paradigms have not evolved much since object oriented programming gained a foothold, and the same

paradigms are present one way or another in many widely used languages. This document will thus give you an overview of the Ada language using analogies to C++ and Java — these are the languages you're already likely to know. No prior knowledge of Ada is assumed. If you are working on an Ada project now and need more background, if you are interested in learning to program in Ada, or if you need to perform an assessment of possible languages to be used for a new development, this guide is for you.

BASICS

Ada implements the vast majority of programming concepts that you're accustomed to in C++ and Java: classes, inheritance, templates (generics), etc. Its syntax might seem peculiar, though. It's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a simpler language — its syntax favors making it easier to conceptualize and read program code, rather than making it faster to write in a cleverly condensed manner. For example, full words like **begin** and **end** are used in place of curly braces. Conditions are written using **if**, **then**, **elsif**, **else**, and **end if**. Ada's assignment operator does not double as an expression, smoothly eliminating any frustration that could be caused by `=` being used where `==` should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C++ and Java. There is no equivalent of `/* ... */` block comments in Ada; use multiple `--` lines instead.

Ada compilers are stricter with type and range checking than most C++ and Java programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding more creatively, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, dynamic checks (such as array bounds checks) provide verification that could not be done at compile time. Dynamic checks are performed at run time, similar to what is done in Java.

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same; likewise **begin**, **BEGIN**, **Begin**, etc. Language-specific characters, such as accents, Greek or Russian letters, and Asian alphabets, are acceptable to use. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

Ada is designed to be portable. Ada compilers must follow a precisely defined international (ISO) standard language specification with clearly documented areas of vendor freedom where the behavior depends on the implementation. It's possible, then, to write an implementation-independent application in Ada and to make sure it will have the same effect across platforms and compilers.

Ada is truly a general purpose, multiple paradigm language that allows the programmer to employ or avoid features like run-time contract checking, tasking, object oriented programming, and generics. Efficiently programmed Ada is employed in device drivers, interrupt handlers, and other low-level functions. It may be found today in devices with tight limits on processing speed, memory, and power consumption. But the language is also used for programming larger interconnected systems running on workstations, servers, and supercomputers.

COMPILATION UNIT STRUCTURE

C++ programming style usually promotes the use of two distinct files: header files used to define specifications (*.h**, *.hxx*, *.hpp*), and implementation files which contain the executable code (*.c*, *.cxx*, *.cpp*). However, the distinction between specification and implementation is not enforced by the compiler and may need to be worked around in order to implement, for example, inlining or templates.

Java compilers expect both the implementation and specification to be in the same *.java* file. (Yes, design patterns allow using interfaces to separate specification from implementation to a certain extent, but this is outside of the scope of this description.)

Ada is superficially similar to the C++ case: Ada compilation units are generally split into two parts, the specification and the body. However, what goes into those files is more predictable for both the compiler and for the programmer. With GNAT, compilation units are stored in files with a *.ads* extension for specifications and with a *.adb* extension for implementations.

Without further ado, we present the famous "Hello World" in three languages:

[Ada]

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Main is
begin
    Put_Line ("Hello World");
end Main;
```

[C++]

```
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]) {
    cout << "Hello World" << endl;
}
```

[Java]

```
public class Main {
    public static void main(String [] argv) {
        System.out.println ("Hello World");
    }
}
```

The first line of Ada we see is the **with** clause, declaring that the unit (in this case, the Main subprogram) will require the services of the package *Ada.Text_IO*. This is different from how **#include** works in C++ in that it does not, in a logical sense, copy/paste the code of *Ada.Text_IO* into Main. The **with** clause directs the compiler to make the public

interface of the `Ada.Text_IO` package visible to code in the unit (here `Main`) containing the **with** clause. Note that this construct does not have a direct analog in Java, where the entire `CLASSPATH` is always accessible. Also, the name `Main` for the main subprogram was chosen for consistency with C++ and Java style but in Ada the name can be whatever the programmer chooses.

The **use** clause is the equivalent of **using namespace** in C++, or **import** in Java (though it wasn't necessary to use **import** in the Java example above). It allows you to omit the full package name when referring to **with**'ed units. Without the **use** clause, any reference to `Ada.Text_IO` items would have had to be fully qualified with the package name. The `Put_Line` line would then have read:

```
Ada.Text_IO.Put_Line ("Hello World");
```

The word "package" has different meanings in Ada and Java. In Java, a package is used as a namespace for classes. In Ada, it's often a compilation unit. As a result Ada tends to have many more packages than Java. Ada package specifications ("package specs" for short) have the following structure:

```
package Package_Name is
    -- public declarations
private
    -- private declarations
end Package_Name;
```

The implementation in a package body (written in a `.adb` file) has the structure:

```
package body Package_Name is
    -- implementation
end Package_Name;
```

The **private** reserved word is used to mark the start of the private portion of a package spec. By splitting the package spec into private and public parts, it is possible to make an entity available for use while hiding its implementation. For instance, a common use is declaring a **record** (Ada's **struct**) whose fields are only visible to its package and not to the caller. This allows the caller to refer to objects of that type, but not to change any of its contents directly.

The package body contains implementation code, and is only accessible to outside code through declarations in the package spec.

An entity declared in the private part of a package in Ada is roughly equivalent to a protected member of a C++ or Java class. An entity declared in the body of an Ada package is roughly equivalent to a private member of a C++ or Java class.

STATEMENTS, DECLARATIONS, AND CONTROL STRUCTURES

48.1 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[Ada]

```
--  
--  Ada program to declare and modify Integers  
--  
procedure Main is  
  --  Variable declarations  
  A, B : Integer := 0;  
  C    : Integer := 100;  
  D    : Integer;  
begin  
  --  Ada uses a regular assignment statement for incrementation.  
  A := A + 1;  
  
  --  Regular addition  
  D := A + B + C;  
end Main;
```

[C++]

```
/*  
 *  C++ program to declare and modify ints  
 */  
int main(int argc, const char* argv[]) {  
  //  Variable declarations  
  int a = 0, b = 0, c = 100, d;  
  
  //  C++ shorthand for incrementation  
  a++;  
  
  //  Regular addition  
  d = a + b + c;  
}
```

[Java]

```
/*  
 *  Java program to declare and modify ints  
 */  
public class Main {  
  public static void main(String [] argv) {
```

(continues on next page)

(continued from previous page)

```
// Variable declarations
int a = 0, b = 0, c = 100, d;

// Java shorthand for incrementation
a++;

// Regular addition
d = a + b + c;
}
}
```

Statements are terminated by semicolons in all three languages. In Ada, blocks of code are surrounded by the reserved words **begin** and **end** rather than by curly braces. We can use both multi-line and single-line comment styles in the C++ and Java code, and only single-line comments in the Ada code.

Ada requires variable declarations to be made in a specific area called the *declarative part*, seen here before the **begin** keyword. Variable declarations start with the identifier in Ada, as opposed to starting with the type as in C++ and Java (also note Ada's use of the **:** separator). Specifying initializers is different as well: in Ada an initialization expression can apply to multiple variables (but will be evaluated separately for each), whereas in C++ and Java each variable is initialized individually. In all three languages, if you use a function as an initializer and that function returns different values on every invocation, each variable will get initialized to a different value.

Let's move on to the imperative statements. Ada does not provide **++** or **--** shorthand expressions for increment/decrement operations; it is necessary to use a full assignment statement. The **:=** symbol is used in Ada to perform value assignment. Unlike C++'s and Java's **=** symbol, **:=** can not be used as part of an expression. So, a statement like **A := B := C;** doesn't make sense to an Ada compiler, and neither does a clause like **if A := B then . . .**. Both are compile-time errors.

You can nest a block of code within an outer block if you want to create an inner scope:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
    Put_Line ("Before the inner block");

    declare
        Alpha : Integer := 0;
    begin
        Alpha := Alpha + 1;
        Put_Line ("Now inside the inner block");
    end;

    Put_Line ("After the inner block");
end Main;
```

It is OK to have an empty declarative part or to omit the declarative part entirely — just start the inner block with **begin** if you have no declarations to make. However it is not OK to have an empty sequence of statements. You must at least provide a **null;** statement, which does nothing and indicates that the omission of statements is intentional.

48.2 Conditions

The use of the **if** statement:

[Ada]

```
if Variable > 0 then
  Put_Line (" > 0 ");
elsif Variable < 0 then
  Put_Line (" < 0 ");
else
  Put_Line (" = 0 ");
end if;
```

[C++]

```
if (Variable > 0)
  cout << " > 0 " << endl;
else if (Variable < 0)
  cout << " < 0 " << endl;
else
  cout << " = 0 " << endl;
```

[Java]

```
if (Variable > 0)
  System.out.println (" > 0 ");
else if (Variable < 0)
  System.out.println (" < 0 ");
else
  System.out.println (" = 0 ");
```

In Ada, everything that appears between the **if** and **then** keywords is the conditional expression — no parentheses required. Comparison operators are the same, except for equality (=) and inequality (/=). The English words **not**, **and**, and **or** replace the symbols **!**, **&**, and **|**, respectively, for performing boolean operations.

It's more customary to use **&&** and **||** in C++ and Java than **&** and **|** when writing boolean expressions. The difference is that **&&** and **||** are short-circuit operators, which evaluate terms only as necessary, and **&** and **|** will unconditionally evaluate all terms. In Ada, **and** and **or** will evaluate all terms; **and then** and **or else** direct the compiler to employ short circuit evaluation.

Here are what switch/case statements look like:

[Ada]

```
case Variable is
  when 0 =>
    Put_Line ("Zero");
  when 1 .. 9 =>
    Put_Line ("Positive Digit");
  when 10 | 12 | 14 | 16 | 18 =>
    Put_Line ("Even Number between 10 and 18");
  when others =>
    Put_Line ("Something else");
end case;
```

[C++]

```
switch (Variable) {
  case 0:
```

(continues on next page)

(continued from previous page)

```

    cout << "Zero" << endl;
    break;
case 1: case 2: case 3: case 4: case 5:
case 6: case 7: case 8: case 9:
    cout << "Positive Digit" << endl;
    break;
case 10: case 12: case 14: case 16: case 18:
    cout << "Even Number between 10 and 18" << endl;
    break;
default:
    cout << "Something else";
}

```

[Java]

```

switch (Variable) {
    case 0:
        System.out.println ("Zero");
        break;
    case 1: case 2: case 3: case 4: case 5:
    case 6: case 7: case 8: case 9:
        System.out.println ("Positive Digit");
        break;
    case 10: case 12: case 14: case 16: case 18:
        System.out.println ("Even Number between 10 and 18");
        break;
    default:
        System.out.println ("Something else");
}

```

In Ada, the **case** and **end case** lines surround the whole case statement, and each case starts with **when**. So, when programming in Ada, replace **switch** with **case**, and replace **case** with **when**.

Case statements in Ada require the use of discrete types (integers or enumeration types), and require all possible cases to be covered by **when** statements. If not all the cases are handled, or if duplicate cases exist, the program will not compile. The default case, **default**: in C++ and Java, can be specified using **when others =>** in Ada.

In Ada, the **break** instruction is implicit and program execution will never fall through to subsequent cases. In order to combine cases, you can specify ranges using **..** and enumerate disjoint values using **|** which neatly replaces the multiple **case** statements seen in the C++ and Java versions.

48.3 Loops

In Ada, loops always start with the **loop** reserved word and end with **end loop**. To leave the loop, use **exit** — the C++ and Java equivalent being **break**. This statement can specify a terminating condition using the **exit when** syntax. The **loop** opening the block can be preceded by a **while** or a **for**.

The **while** loop is the simplest one, and is very similar across all three languages:

[Ada]

```

while Variable < 10_000 loop
    Variable := Variable * 2;
end loop;

```

[C++]

```
while (Variable < 10000) {
    Variable = Variable * 2;
}
```

[Java]

```
while (Variable < 10000) {
    Variable = Variable * 2;
}
```

Ada's **for** loop, however, is quite different from that in C++ and Java. It always increments or decrements a loop index within a discrete range. The loop index (or "loop parameter" in Ada parlance) is local to the scope of the loop and is implicitly incremented or decremented at each iteration of the loop statements; the program cannot directly modify its value. The type of the loop parameter is derived from the range. The range is always given in ascending order even if the loop iterates in descending order. If the starting bound is greater than the ending bound, the interval is considered to be empty and the loop contents will not be executed. To specify a loop iteration in decreasing order, use the **reverse** reserved word. Here are examples of loops going in both directions:

[Ada]

```
-- Outputs 0, 1, 2, ..., 9
for Variable in 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;

-- Outputs 9, 8, 7, ..., 0
for Variable in reverse 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;
```

[C++]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
    cout << Variable << endl;
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
    cout << Variable << endl;
}
```

[Java]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
    System.out.println (Variable);
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
    System.out.println (Variable);
}
```

Ada uses the **Integer** type's **'Image** attribute to convert a numerical value to a String. There is no implicit conversion between **Integer** and **String** as there is in C++ and Java. We'll have a more in-depth look at such attributes later on.

It's easy to express iteration over the contents of a container (for instance, an array, a list, or a map) in Ada and Java. For example, assuming that `Int_List` is defined as an array of

Integer values, you can use:

[Ada]

```
for I of Int_List loop
  Put_Line (Integer'Image (I));
end loop;
```

[Java]

```
for (int i : Int_List) {
  System.out.println (i);
}
```

TYPE SYSTEM

49.1 Strong Typing

One of the main characteristics of Ada is its strong typing (i.e., relative absence of implicit type conversions). This may take some getting used to. For example, you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

```
procedure Strong_Typing is
  Alpha : Integer := 1;
  Beta  : Integer := 10;
  Result : Float;
begin
  Result := Float (Alpha) / Float (Beta);
end Strong_Typing;
```

[C++]

```
void weakTyping () {
  int  alpha = 1;
  int  beta = 10;
  float result;

  result = alpha / beta;
}
```

[Java]

```
void weakTyping () {
  int  alpha = 1;
  int  beta = 10;
  float result;

  result = alpha / beta;
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from Integer to Float explicit. In fact it significantly changes the behavior of the computation. While the Ada code performs a floating point operation $1.0 / 10.0$ and stores 0.1 in Result, the C++ and Java versions instead store 0.0 in result. This is because the C++ and Java versions perform an integer operation between two integer variables: $1 / 10$ is 0. The result of the integer division is then converted to a float and stored. Errors of this sort can be very hard to locate in complex

pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to **Float**:

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

In Ada, a floating point literal must be written with both an integral and decimal part. `10` is not a valid literal for a floating point value, while `10.0` is.

49.2 Language-Defined Types

The principal scalar types predefined by Ada are **Integer**, **Float**, **Boolean**, and **Character**. These correspond to **int**, **float**, **bool/boolean**, and **char**, respectively. The names for these types are not reserved words; they are regular identifiers.

49.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators `+`, `-`, `*`, and `/` for integer and floating-point types take operands of the same type and return a value of that type.

```
procedure Main is
  type Distance is new Float;
  type Area is new Float;

  D1 : Distance := 2.0;
  D2 : Distance := 3.0;
  A  : Area;
begin
  D1 := D1 + D2;           -- OK
  D1 := D1 + A;           -- NOT OK: incompatible types for "+" operator
  A  := D1 * D2;           -- NOT OK: incompatible types for "!=" assignment
  A  := Area (D1 * D2);    -- OK
end Main;
```

Even though the Distance and Area types above are just **Floats**, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two Distances yields a Distance) and prohibit some useful cases (for example multiplying two Distances should deliver an Area). These situations can be handled through other mechanisms. A predefined operation can be identified as **abstract** to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C++ and Java's **enums**.

[Ada]

```

type Day is
  (Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday);

```

[C++]

```

enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday};

```

[Java]

```

enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday}

```

But even though such enumerations may be implemented using a machine word, at the language level Ada will not confuse the fact that Monday is a Day and is not an **Integer**. You can compare a Day with another Day, though. To specify implementation details like the numeric values that correspond with enumeration values in C++ you include them in the original **enum** statement:

[C++]

```

enum Day {
  Monday    = 10,
  Tuesday   = 11,
  Wednesday = 12,
  Thursday  = 13,
  Friday    = 14,
  Saturday  = 15,
  Sunday    = 16};

```

But in Ada you must use both a type definition for Day as well as a separate *representation clause* for it like:

[Ada]

```

for Day use
  (Monday    => 10,
   Tuesday   => 11,
   Wednesday => 12,
   Thursday  => 13,
   Friday    => 14,
   Saturday  => 15,
   Sunday    => 16);

```


49.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the **range** reserved word, for example:

```
procedure Main is
  type Grade is range 0 .. 100;

  G1, G2 : Grade;
  N       : Integer;
begin
  ...           -- Initialization of N
  G1 := 80;     -- OK
  G1 := N;      -- Illegal (type mismatch)
  G1 := Grade (N); -- Legal, run-time range check
  G2 := G1 + 10; -- Legal, run-time range check
  G1 := (G1 + G2)/2; -- Legal, run-time range check
end Main;
```

In the above example, Grade is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to G1 is correct and will not raise a run-time exception. Assigning N to G1 is illegal since Grade is a different type than **Integer**. Converting N to Grade makes the assignment legal, and a range check on the conversion confirms that the value is within 0 .. 100. Assigning G1+10 to G2 is legal since + for Grade returns a Grade (note that the literal 10 is interpreted as a Grade value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression G1 + G2 may be outside the range of Grade, but the final result will be in range. Nevertheless, depending on the representation chosen for Grade, the addition may overflow. If the compiler represents Grade values as signed 8-bit integers (i.e., machine numbers in the range -128 .. 127) then the sum G1+G2 may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

```
G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
```

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the Day example, one can write:

```
type Business_Day is new Day range Monday .. Friday;
type Weekend_Day is new Day range Saturday .. Sunday;
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind *subtypes* in Ada. A subtype is a type with optional additional constraints. For example:

```
subtype Business_Day is Day range Monday .. Friday;
subtype Weekend_Day is Day range Saturday .. Sunday;
subtype Dice_Throw is Integer range 1 .. 6;
```

These declarations don't create new types, just new names for constrained ranges of their

base types.

49.5 Generalized Type Contracts: Subtype Predicates

Range checks are a special form of type contracts; a more general method is provided by Ada subtype predicates, introduced in Ada 2012. A subtype predicate is a boolean expression defining conditions that are required for a given type or subtype. For example, the `Dice_Throw` subtype shown above can be defined in the following way:

```
subtype Dice_Throw is Integer
  with Dynamic_Predicate => Dice_Throw in 1 .. 6;
```

The clause beginning with `with` introduces an Ada *aspect*, which is additional information provided for declared entities such as types and subtypes. The `Dynamic_Predicate` aspect is the most general form. Within the predicate expression, the name of the (sub)type refers to the current value of the (sub)type. The predicate is checked on assignment, parameter passing, and in several other contexts. There is a `Static_Predicate` form which introduces some optimization and constrains on the form of these predicates, outside of the scope of this document.

Of course, predicates are useful beyond just expressing ranges. They can be used to represent types with arbitrary constraints, in particular types with discontinuities, for example:

```
type Not_Null is new Integer
  with Dynamic_Predicate => Not_Null /= 0;

type Even is new Integer
  with Dynamic_Predicate => Even mod 2 = 0;
```

49.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The `'Image` and `'Value` attributes allow you to transform a scalar value into a `String` and vice-versa. For example:

```
declare
  A : Integer := 99;
begin
  Put_Line (Integer'Image (A));
  A := Integer'Value ("99");
end;
```

Certain attributes are provided only for certain kinds of types. For example, the `'Val` and `'Pos` attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

```
declare
  C : Character := 'a';
begin
  C := Character'Val (Character'Pos (C) + 1);
end;
```

A more concise way to get the next value in Ada is to use the 'Succ attribute:

```
declare
  C : Character := 'a';
begin
  C := Character'Succ (C);
end;
```

You can get the previous value using the 'Pred attribute. Here is the equivalent in C++ and Java:

[C++]

```
char c = 'a';
c++;
```

[Java]

```
char c = 'a';
c++;
```

Other interesting examples are the 'First and 'Last attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, Integer'First returns -2^{31} and Integer'Last returns $2^{31} - 1$.

49.7 Arrays and Strings

C++ arrays are pointers with offsets, but the same is not the case for Ada and Java. Arrays in the latter two languages are not interchangeable with operations on pointers, and array types are considered first-class citizens. Arrays in Ada have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from 'a' to 'z':

[Ada]

```
declare
  type Arr_Type is array (Integer range <>) of Character;
  Arr : Arr_Type (1 .. 26);
  C : Character := 'a';
begin
  for I in Arr'Range loop
    Arr (I) := C;
    C := Character'Succ (C);
  end loop;
end;
```

[C++]

```
char Arr [26];
char C = 'a';

for (int I = 0; I < 26; ++I) {
  Arr [I] = C;
  C = C + 1;
}
```

[Java]

```
char [] Arr = new char [26];
char C = 'a';

for (int I = 0; I < Arr.length; ++I) {
    Arr [I] = C;
    C = C + 1;
}
```

In C++ and Java, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of Integer as the index type, but does not provide any constraints (use <>, pronounced *box*, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called **'Range'**. In our example, Arr**'Range'** can also be expressed as Arr**'First'** .. Arr**'Last'**; both expressions will resolve to 1 .. 26. So the **'Range'** attribute supplies the bounds for our **for** loop. There is no risk of stating either of the bounds incorrectly, as one might do in C++ where `I <= 26` may be specified as the end-of-loop condition.

As in C++, Ada **Strings** are arrays of **Characters**. The C++ or Java String class is the equivalent of the Ada type `Ada.Strings.Unbounded_String` which offers additional capabilities in exchange for some overhead. Ada strings, importantly, are not delimited with the special character `'\0'` like they are in C++. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined **String** type is very straightforward to use:

```
My_String : String (1 .. 26);
```

Unlike C++ and Java, Ada does not offer escape sequences such as `'\n'`. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, &). Here for example, is how to initialize a line of text ending with a new line:

```
My_String : String := "This is a line with a end of line" & ASCII.LF;
```

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of My_String's bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C++ or Java, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer;
    A1 : Arr_Type (1 .. 2);
    A2 : Arr_Type (1 .. 2);
begin
    A1 (1) := 0;
    A1 (2) := 1;

    A2 := A1;
end;
```

[C++]

```
int A1 [2];
int A2 [2];
```

(continues on next page)

(continued from previous page)

```
A1 [0] = 0;
A1 [1] = 1;

for (int i = 0; i < 2; ++i) {
    A2 [i] = A1 [i];
}
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

A1 [0] = 0;
A1 [1] = 1;

A2 = Arrays.copyOf(A1, A1.length);
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer;
    A1 : Arr_Type (1 .. 10);
    A2 : Arr_Type (1 .. 5);
begin
    A2 (1 .. 3) := A1 (4 .. 6);
end;
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer;
    A1 : Arr_Type (1 .. 2);
    A2 : Arr_Type (1 .. 2);
begin
    if A1 = A2 then
```

[C++]

```
int A1 [2];
int A2 [2];

bool eq = true;

for (int i = 0; i < 2; ++i) {
    if (A1 [i] != A2 [i]) {
        eq = false;
    }
}

if (eq) {
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

if (Arrays.equals (A1, A2)) {
```

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the **others** expression to define a default value for all the unspecified array elements. Therefore, you can write:

```
declare
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
  A2 : Arr_Type (-2 .. 42) := (others => 0);
begin
  A1 := (1, 2, 3, others => 10);

  -- use a slice to assign A2 elements 11 .. 19 to 1
  A2 (11 .. 19) := (others => 1);
end;
```

49.8 Heterogeneous Data Structures

In Ada, there's no distinction between **struct** and **class** as there is in C++. All heterogeneous data structures are **records**. Here are some simple records:

[Ada]

```
declare
  type R is record
    A, B : Integer;
    C    : Float;
  end record;

  V : R;
begin
  V.A := 0;
end;
```

[C++]

```
struct R {
  int A, B;
  float C;
};

R V;
V.A = 0;
```

[Java]

```
class R {
  public int A, B;
  public float C;
}

R V = new R ();
V.A = 0;
```

Ada allows specification of default values for fields just like C++ and Java. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by **others** => <> to specify that fields not listed will take their default values. For example:

```
type R is record
  A, B : Integer := 0;
  C    : Float  := 0.0;
end record;

V1 : R := (1, 2, 1.0);
V2 : R := (A => 1, B => 2, C => 1.0);
V3 : R := (C => 1.0, A => 1, B => 2);
V4 : R := (C => 1.0, others => <>);
```

49.9 Pointers

Pointers, references, and access types differ in significant ways across the languages that we are examining. In C++, pointers are integral to a basic understanding of the language, from array manipulation to proper declaration and use of function parameters. In Java, direct pointer manipulation is abstracted by the Java runtime. And in Ada, direct pointer manipulation is possible, but unlike C++, they are not required for basic usage with arrays and parameter passing.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

```
declare
  type R is record
    A, B : Integer;
  end record;

  V1, V2 : R;
begin
  V1.A := 0;
  V2 := V1;
  V2.A := 1;
end;
```

[C++]

```
struct R {
  int A, B;
};

R V1, V2;
V1.A = 0;
V2 = V1;
V2.A = 1;
```

[Java]

```
public class R {
  public int A, B;
}

R V1, V2;
V1 = new R ();
```

(continues on next page)

(continued from previous page)

```
V1.A = 0;
V2 = V1;
V2.A = 1;
```

There's a fundamental difference between the Ada and C++ semantics above and the semantics for Java. In Ada and C++, objects are allocated on the stack and are directly accessed. V1 and V2 are two different objects and the assignment statement copies the value of V1 into V2. In Java, V1 and V2 are two *references* to objects of class R. Note that when V1 and V2 are declared, no actual object of class R yet exists in memory: it has to be allocated later with the **new** allocator operator. After the assignment V2 = V1, there's only one R object in memory: the assignment is a reference assignment, not a value assignment. At the end of the Java code, V1 and V2 are two references to the same objects and the V2.A = 1 statement changes the field of that one object, while in the Ada and the C++ case V1 and V2 are two distinct objects.

To obtain similar behavior in Ada, you can use pointers. It can be done through Ada's *access type*:

[Ada]

```
declare
  type R is record
    A, B : Integer;
  end record;
  type R_Access is access R;

  V1 : R_Access;
  V2 : R_Access;
begin
  V1 := new R;
  V1.A := 0;
  V2 := V1;
  V2.A := 1;
end;
```

[C++]

```
struct R {
  int A, B;
};

R * V1, * V2;
V1 = new R ();
V1->A = 0;
V2 = V1;
V2->A = 0;
```

For those coming from the Java world: there's no garbage collector in Ada, so objects allocated by the **new** operator need to be expressly freed.

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append **.all**. The equivalent of V1->A in C++ can be written either as V1.A or V1.all.A.

Pointers to scalar objects in Ada and C++ look like:

[Ada]

```
procedure Main is
  type A_Int is access Integer;
  Var : A_Int := new Integer;
```

(continues on next page)

(continued from previous page)

```
begin
  Var.all := 0;
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
  int * Var = new int;
  *Var = 0;
}
```

An initializer can be specified with the allocation by appending '(value)':

```
Var : A_Int := new Integer'(0);
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being **aliased**. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either **access all** (if the referenced object needs to be assigned to) or **access constant** (if the referenced object is a constant). The **'Access** attribute works like the C++ **&** operator to get a pointer to the object, but with a "scope accessibility" check to prevent references to objects that have gone out of scope. For example:

[Ada]

```
type A_Int is access all Integer;
Var : aliased Integer;
Ptr : A_Int := Var'Access;
```

[C++]

```
int Var;
int * Ptr = &Var;
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs — it's called `Ada.Unchecked_Deallocation`. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

```
with Ada.Unchecked_Deallocation;
procedure Main is
  type Integer_Access is access all Integer;
  procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
  My_Pointer : Integer_Access := new Integer;
begin
  Free (My_Pointer);
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
  int * my_pointer = new int;
  delete my_pointer;
}
```

FUNCTIONS AND PROCEDURES

50.1 General Form

Subroutines in C++ and Java are always expressed as functions (methods) which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term "subprogram" to refer to both functions and procedures.

Parameters can be passed in three distinct modes: **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram. **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller. **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant reference in C++). Ada also provides **access** parameters, in effect an explicit pass-by-reference indicator.

In Ada the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). (There are some exceptions to the "in general". For example, parameters of scalar types are always passed by copy, for all three modes.) C++ has the programmer specify how to pass the parameter, and Java forces primitive type parameters to be passed by copy and all other parameters to be passed by reference. For this reason, a 1:1 mapping between Ada and Java isn't obvious but here's an attempt to show these differences:

[Ada]

```
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer);

function Func (Var : Integer) return Integer;

procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer)
is
begin
  Var2 := Func (Var1);
  Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
  return Var + 1;
end Func;
```

[C++]

```
void Proc
(int Var1,
 int & Var2,
 int & Var3);

int Func (int Var);

void Proc
(int Var1,
 int & Var2,
 int & Var3) {

    Var2 = Func (Var1);
    Var3 = Var3 + 1;
}

int Func (int Var) {
    return Var + 1;
}
```

[Java]

```
public class ProcData {
    public int Var2;
    public int Var3;

    public void Proc (int Var1) {
        Var2 = Func (Var1);
        Var3 = Var3 + 1;
    }

    public static int Func (int Var) {
        return Var + 1;
    }
}
```

The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C++, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared. In Java, it's fine to have the declaration of the subprogram later .

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax, including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

50.2 Overloading

Different subprograms may share the same name; this is called "overloading." As long as the subprogram signatures (subprogram name, parameter types, and return types) are different, the compiler will be able to resolve the calls to the proper destinations. For example:

```
function Value (Str : String) return Integer;
function Value (Str : String) return Float;

V : Integer := Value ("8");
```

The Ada compiler knows that an assignment to *V* requires an **Integer**. So, it chooses the *Value* function that returns an **Integer** to satisfy this requirement.

Operators in Ada can be treated as functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To express an operator as a function, enclose it in quotes:

[Ada]

```
function "=" (Left : Day; Right : Integer) return Boolean;
```

[C++]

```
bool operator = (Day Left, int Right);
```

50.3 Subprogram Contracts

You can express the expected inputs and outputs of subprograms by specifying subprogram contracts. The compiler can then check for valid conditions to exist when a subprogram is called and can check that the return value makes sense. Ada allows defining contracts in the form of Pre and Post conditions; this facility was introduced in Ada 2012. They look like:

```
function Divide (Left, Right : Float) return Float
  with Pre => Right /= 0.0,
       Post => Divide'Result * Right < Left + 0.0001
          and then Divide'Result * Right > Left - 0.0001;
```

The above example adds a Pre condition, stating that *Right* cannot be equal to 0.0. While the IEEE floating point standard permits divide-by-zero, you may have determined that use of the result could still lead to issues in a particular application. Writing a contract helps to detect this as early as possible. This declaration also provides a Post condition on the result.

Postconditions can also be expressed relative to the value of the input:

```
procedure Increment (V : in out Integer)
  with Pre => V < Integer'Last,
       Post => V = V'Old + 1;
```

V'Old in the postcondition represents the value that *V* had before entering *Increment*.

PACKAGES

51.1 Declaration Protection

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C++. An Ada package contains three parts that, for GNAT, are separated into two files: *.ads* files contain public and private Ada specifications, and *.adb* files contain the implementation, or Ada bodies.

Java doesn't provide any means to cleanly separate the specification of methods from their implementation: they all appear in the same file. You can use interfaces to emulate having separate specifications, but this requires the use of OOP techniques which is not always practical.

Ada and C++ do offer separation between specifications and implementations out of the box, independent of OOP.

```
package Package_Name is
    -- public specifications
private
    -- private specifications
end Package_Name;

package body Package_Name is
    -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. The **private** keyword splits the package spec into "public" and "private" parts. That is somewhat analogous to C++'s partitioning of the class construct into different sections with different visibility properties. In Java, the encapsulation has to be done field by field, but in Ada the entire definition of a type can be hidden. For example:

```
package Types is
    type Type_1 is private;
    type Type_2 is private;
    type Type_3 is private;
    procedure P (X : Type_1);
    ...
private
    procedure Q (Y : Type_1);
    type Type_1 is new Integer range 1 .. 1000;
    type Type_2 is array (Integer range 1 .. 1000) of Integer;
    type Type_3 is record
        A, B : Integer;
    end record;
end Types;
```

Subprograms declared above the **private** separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implemen-

tation, has access to both parts.

51.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

```
-- root-child.ads

package Root.Child is
    -- package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
    -- package body goes here
end Root.Child;
```

Here, `Root.Child` is a child package of `Root`. The public part of `Root.Child` has access to the public part of `Root`. The private part of `Child` has access to the private part of `Root`, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

51.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a `with` clause that references the package, which is similar to the C++ `#include` directive. Visibility is implicit in Java: you can always access all classes located in your `CLASSPATH`. After a `with` clause, entities need to be prefixed by the name of their package, like a C++ namespace or a Java package. This prefix can be omitted if a `use` clause is employed, similar to a C++ `using namespace` or a Java `import`.

[Ada]

```
-- pck.ads

package Pck is
    My_Glob : Integer;
end Pck;

-- main.adb

with Pck;

procedure Main is
begin
    Pck.My_Glob := 0;
end Main;
```

[C++]

```
// pck.h

namespace pck {
    extern int myGlob;
}
```

(continues on next page)

(continued from previous page)

```
// pck.cpp

namespace pck {
    int myGlob;
}

// main.cpp

#include "pck.h"

int main (int argc, char ** argv) {
    pck::myGlob = 0;
}
```

[Java]

```
// Globals.java

package pck;

public class Globals {
    public static int myGlob;
}

// Main.java

public class Main {
    public static void main (String [] argv) {
        pck.Globals.myGlob = 0;
    }
}
```


CLASSES AND OBJECT ORIENTED PROGRAMMING

52.1 Primitive Subprograms

Primitive subprograms in Ada are basically the subprograms that are eligible for inheritance / derivation. They are the equivalent of C++ member functions and Java instance methods. While in C++ and Java these subprograms are located within the nested scope of the type, in Ada they are simply declared in the same scope as the type. There's no syntactic indication that a subprogram is a primitive of a type.

The way to determine whether P is a primitive of a type T is if

1. it is declared in the same scope as T, and
2. it contains at least one parameter of type T, or returns a result of type T.

In C++ or Java, the self reference **this** is implicitly declared. It may need to be explicitly stated in certain situations, but usually it's omitted. In Ada the self-reference, called the *controlling parameter*, must be explicitly specified in the subprogram parameter list. While it can be any parameter in the profile with any name, we'll focus on the typical case where the first parameter is used as the self parameter. Having the controlling parameter listed first also enables the use of OOP prefix notation which is convenient.

A **class** in C++ or Java corresponds to a **tagged type** in Ada. Here's an example of the declaration of an Ada tagged type with two parameters and some dispatching and non-dispatching primitives, with equivalent examples in C++ and Java:

[Ada]

```
type T is tagged record
  V, W : Integer;
end record;

type T_Access is access all T;

function F (V : T) return Integer;

procedure P1 (V : access T);

procedure P2 (V : T_Access);
```

[C++]

```
class T {
public:
  int V, W;

  int F ();

  void P1 ();
};
```

(continues on next page)

(continued from previous page)

```
void P2 (T * v);
```

[Java]

```
public class T {  
    public int V, W;  
  
    public int F () {};  
  
    public void P1 () {};  
  
    public static void P2 (T v) {};  
}
```

Note that P2 is not a primitive of T — it does not have any parameters of type T. Its parameter is of type T_Access, which is a different type.

Once declared, primitives can be called like any subprogram with every necessary parameter specified, or called using prefix notation. For example:

[Ada]

```
declare  
    V : T;  
begin  
    V.P1;  
end;
```

[C++]

```
{  
    T v;  
    v.P1 ();  
}
```

[Java]

```
{  
    T v = new T ();  
    v.P1 ();  
}
```

52.2 Derivation and Dynamic Dispatch

Despite the syntactic differences, derivation in Ada is similar to derivation (inheritance) in C++ or Java. For example, here is a type hierarchy where a child class overrides a method and adds a new method:

[Ada]

```
type Root is tagged record  
    F1 : Integer;  
end record;  
  
procedure Method_1 (Self : Root);  
  
type Child is new Root with record  
    F2 : Integer;
```

(continues on next page)

(continued from previous page)

```

end record;

overriding
procedure Method_1 (Self : Child);

procedure Method_2 (Self : Child);

```

[C++]

```

class Root {
    public:
        int f1;
        virtual void method1 ();
};

class Child : public Root {
    public:
        int f2;
        virtual void method1 ();
        virtual void method2 ();
};

```

[Java]

```

public class Root {
    public int f1;
    public void method1 ();
}

public class Child extends Root {
    public int f2;
    @Override
    public void method1 ();
    public void method2 ();
}

```

Like Java, Ada primitives on tagged types are always subject to dispatching; there is no need to mark them **virtual**. Also like Java, there's an optional keyword **overriding** to ensure that a method is indeed overriding something from the parent type.

Unlike many other OOP languages, Ada differentiates between a reference to a specific tagged type, and a reference to an entire tagged type hierarchy. While `Root` is used to mean a specific type, `Root'Class` — a class-wide type — refers to either that type or any of its descendants. A method using a parameter of such a type cannot be overridden, and must be passed a parameter whose type is of any of `Root`'s descendants (including `Root` itself).

Next, we'll take a look at how each language finds the appropriate method to call within an OO class hierarchy; that is, their dispatching rules. In Java, calls to non-private instance methods are always dispatching. The only case where static selection of an instance method is possible is when calling from a method to the **super** version.

In C++, by default, calls to virtual methods are always dispatching. One common mistake is to use a by-copy parameter hoping that dispatching will reach the real object. For example:

```

void proc (Root p) {
    p.method1 ();
}

Root * v = new Child ();

proc (*v);

```

In the above code, `p.method1()` will not dispatch. The call to `proc` makes a copy of the `Root` part of `v`, so inside `proc`, `p.method1()` refers to the `method1()` of the root object. The intended behavior may be specified by using a reference instead of a copy:

```
void proc (Root & p) {
    p.method1 ();
}

Root * v = new Child ();

proc (*v);
```

In Ada, tagged types are always passed by reference but dispatching only occurs on class-wide types. The following Ada code is equivalent to the latter C++ example:

```
declare
    procedure Proc (P : Root'Class) is
    begin
        P.Method_1;
    end;

    type Root_Access is access all Root'Class;
    V : Root_Access := new Child;
begin
    Proc (V.all);
end;
```

Dispatching from within primitives can get tricky. Let's consider a call to `Method_1` in the implementation of `Method_2`. The first implementation that might come to mind is:

```
procedure Method_2 (P : Root) is
begin
    P.Method_1;
end;
```

However, `Method_2` is called with a parameter that is of the definite type `Root`. More precisely, it is a definite view of a child. So, this call is not dispatching; it will always call `Method_1` of `Root` even if the object passed is a child of `Root`. To fix this, a view conversion is necessary:

```
procedure Method_2 (P : Root) is
begin
    Root'Class (P).Method_1;
end;
```

This is called "redispatching." Be careful, because this is the most common mistake made in Ada when using OOP. In addition, it's possible to convert from a class wide view to a definite view, and to select a given primitive, like in C++:

[Ada]

```
procedure Proc (P : Root'Class) is
begin
    Root (P).Method_1;
end;
```

[C++]

```
void proc (Root & p) {
    p.Root::method1 ();
}
```

52.3 Constructors and Destructors

Ada does not have constructors and destructors in quite the same way as C++ and Java, but there is analogous functionality in Ada in the form of default initialization and finalization.

Default initialization may be specified for a record component and will occur if a variable of the record type is not assigned a value at initialization. For example:

```
type T is tagged record
  F : Integer := Compute_Default_F;
end record;

function Compute_Default_F return Integer is
begin
  Put_Line ("Compute");
  return 0;
end Compute_Default_F;

V1 : T;
V2 : T := (F => 0);
```

In the declaration of V1, T.F receives a value computed by the subprogram `Compute_Default_F`. This is part of the default initialization. V2 is initialized manually and thus will not use the default initialization.

For additional expressive power, Ada provides a type called `Ada.Finalization.Controlled` from which you can derive your own type. Then, by overriding the `Initialize` procedure you can create a constructor for the type:

```
type T is new Ada.Finalization.Controlled with record
  F : Integer;
end record;

procedure Initialize (Self : in out T) is
begin
  Put_Line ("Compute");
  Self.F := 0;
end Initialize;

V1 : T;
V2 : T := (F => 0);
```

Again, this default initialization subprogram is only called for V1; V2 is initialized manually. Furthermore, unlike a C++ or Java constructor, `Initialize` is a normal subprogram and does not perform any additional initialization such as calling the parent's initialization routines.

When deriving from `Controlled`, it's also possible to override the subprogram `Finalize`, which is like a destructor and is called for object finalization. Like `Initialize`, this is a regular subprogram. Do not expect any other finalizers to be automatically invoked for you.

Controlled types also provide functionality that essentially allows overriding the meaning of the assignment operation, and are useful for defining types that manage their own storage reclamation (for example, implementing a reference count reclamation strategy).

52.4 Encapsulation

While done at the class level for C++ and Java, Ada encapsulation occurs at the package level and targets all entities of the language, as opposed to only methods and attributes. For example:

[Ada]

```
package Pck is
  type T is tagged private;
  procedure Method1 (V : T);
private
  type T is tagged record
    F1, F2 : Integer;
  end record;
  procedure Method2 (V : T);
end Pck;
```

[C++]

```
class T {
public:
  virtual void method1 ();
protected:
  int f1, f2;
  virtual void method2 ();
};
```

[Java]

```
public class T {
  public void method1 ();
  protected int f1, f2;
  protected void method2 ();
}
```

The C++ and Java code's use of **protected** and the Ada code's use of **private** here demonstrates how to map these concepts between languages. Indeed, the private part of an Ada child package would have visibility of the private part of its parents, mimicking the notion of **protected**. Only entities declared in the package body are completely isolated from access.

52.5 Abstract Types and Interfaces

Ada, C++ and Java all offer similar functionality in terms of abstract classes, or pure virtual classes. It is necessary in Ada and Java to explicitly specify whether a tagged type or class is **abstract**, whereas in C++ the presence of a pure virtual function implicitly makes the class an abstract base class. For example:

[Ada]

```
package P is
  type T is abstract tagged private;

  procedure Method (Self : T) is abstract;
private
  type T is abstract tagged record
    F1, F2 : Integer;
  end record;

end P;
```

[C++]

```
class T {
  public:
    virtual void method () = 0;
  protected:
    int f1, f2;
};
```

[Java]

```
public abstract class T {
  public abstract void method1 ();
  protected int f1, f2;
};
```

All abstract methods must be implemented when implementing a concrete type based on an abstract type.

Ada doesn't offer multiple inheritance the way C++ does, but it does support a Java-like notion of interfaces. An interface is like a C++ pure virtual class with no attributes and only abstract members. While an Ada tagged type can inherit from at most one tagged type, it may implement multiple interfaces. For example:

[Ada]

```
type Root is tagged record
  F1 : Integer;
end record;
procedure M1 (Self : Root);

type I1 is interface;
procedure M2 (Self : I1) is abstract;

type I2 is interface;
procedure M3 (Self : I2) is abstract;

type Child is new Root and I1 and I2 with record
  F2 : Integer;
end record;

-- M1 implicitly inherited by Child
procedure M2 (Self : Child);
procedure M3 (Self : Child);
```

[C++]

```
class Root {
  public:
    virtual void M1();
    int f1;
};

class I1 {
  public:
    virtual void M2 () = 0;
};

class I2 {
  public:
    virtual void M3 () = 0;
};

class Child : public Root, I1, I2 {
```

(continues on next page)

(continued from previous page)

```

public:
    int f2;
    virtual void M2 ();
    virtual void M3 ();
};

```

[Java]

```

public class Root {
    public void M1();
    public int f1;
}

public interface I1 {
    public void M2 ();
}

public interface I2 {
    public void M3 ();
}

public class Child extends Root implements I1, I2 {
    public int f2;
    public void M2 ();
    public void M3 ();
}

```

52.6 Invariants

Any private type in Ada may be associated with a `Type_Invariant` contract. An invariant is a property of a type that must always be true after the return from of any of its primitive subprograms. (The invariant might not be maintained during the execution of the primitive subprograms, but will be true after the return.) Let's take the following example:

```

package Int_List_Pkg is

    type Int_List (Max_Length : Natural) is private
        with Type_Invariant => Is_Sorted (Int_List);

    function Is_Sorted (List : Int_List) return Boolean;

    type Int_Array is array (Positive range <>) of Integer;

    function To_Int_List (Ints : Int_Array) return Int_List;

    function To_Int_Array (List : Int_List) return Int_Array;

    function "&" (Left, Right : Int_List) return Int_List;

    ... -- Other subprograms
private

    type Int_List (Max_Length : Natural) is record
        Length : Natural;
        Data : Int_Array (1..Max_Length);
    end record;

    function Is_Sorted (List : Int_List) return Boolean is

```

(continues on next page)

(continued from previous page)

```

        (for all I in List.Data'First .. List.Length-1 =>
         List.Data (I) <= List.Data (I+1));

end Int_List_Pkg;

package body Int_List_Pkg is

    procedure Sort (Ints : in out Int_Array) is
    begin
        ... Your favorite sorting algorithm
    end Sort;

    function To_Int_List (Ints : Int_Array) return Int_List is
        List : Int_List :=
            (Max_Length => Ints'Length,
             Length      => Ints'Length,
             Data        => Ints);
    begin
        Sort (List.Data);
        return List;
    end To_Int_List;

    function To_Int_Array (List : Int_List) return Int_Array is
    begin
        return List.Data;
    end To_Int_Array;

    function "&" (Left, Right : Int_List) return Int_List is
        Ints : Int_Array := Left.Data & Right.Data;
    begin
        Sort (Ints);
        return To_Int_List (Ints);
    end "&";

    ... -- Other subprograms
end Int_List_Pkg;

```

The `Is_Sorted` function checks that the type stays consistent. It will be called at the exit of every primitive above. It is permissible if the conditions of the invariant aren't met during execution of the primitive. In `To_Int_List` for example, if the source array is not in sorted order, the invariant will not be satisfied at the "begin", but it will be checked at the end.

GENERICIS

Ada, C++, and Java all have support for generics or templates, but on different sets of language entities. A C++ template can be applied to a class or a function. So can a Java generic. An Ada generic can be either a package or a subprogram.

53.1 Generic Subprograms

In this example, we will swap two generic objects. This is possible in Ada and C++ using a temporary variable. In Java, parameters are a copy of a reference value that is passed into the function, so modifying those references in the function scope has no effect from the caller's context. A generic swap method, like the below Ada or C++ examples is not possible in Java, so we will skip the Java version of this example.

[Ada]

```
generic
  type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
  Temp : A_Type := Left;
begin
  Left  := Right;
  Right := Temp;
end Swap;
```

[C++]

```
template <class AType>
AType swap (AType & left, AType & right) {
  AType temp = left;
  left = right;
  right = temp;
}
```

And examples of using these:

[Ada]

```
declare
  type R is record
    F1, F2 : Integer;
  end record;

  procedure Swap_R is new Swap (R);
  A, B : R;
begin
  ...
  Swap_R (A, B);
end;
```

[C++]

```
class R {
    public:
        int f1, f2;
};

R a, b;
...
swap (a, b);
```

The C++ template becomes usable once defined. The Ada generic needs to be explicitly instantiated using a local name and the generic's parameters.

53.2 Generic Packages

Next, we're going to create a generic unit containing data and subprograms. In Java or C++, this is done through a class, while in Ada, it's a *generic package*. The Ada and C++ model is fundamentally different from the Java model. Indeed, upon instantiation, Ada and C++ generic data are duplicated; that is, if they contain global variables (Ada) or static attributes (C++), each instance will have its own copy of the variable, properly typed and independent from the others. In Java, generics are only a mechanism to have the compiler do consistency checks, but all instances are actually sharing the same data where the generic parameters are replaced by *java.lang.Object*. Let's look at the following example:

[Ada]

```
generic
    type T is private;
package Gen is
    type C is tagged record
        V : T;
    end record;

    G : Integer;
end Gen;
```

[C++]

```
template <class T>
class C{
    public:
        T v;
        static int G;
};
```

[Java]

```
public class C <T> {
    public T v;
    public static int G;
}
```

In all three cases, there's an instance variable (v) and a static variable (G). Let's now look at the behavior (and syntax) of these three instantiations:

[Ada]

```
declare
    package I1 is new Gen (Integer);
```

(continues on next page)

(continued from previous page)

```

package I2 is new Gen (Integer);
subtype Str10 is String (1..10);
package I3 is new Gen (Str10);
begin
  I1.G := 0;
  I2.G := 1;
  I3.G := 2;
end;

```

[C++]

```

C <int>::G = 0;
C <int>::G = 1;
C <char *>::G = 2;

```

[Java]

```

C.G = 0;
C.G = 1;
C.G = 2;

```

In the Java case, we access the generic entity directly without using a parametric type. This is because there's really only one instance of C, with each instance sharing the same global variable G. In C++, the instances are implicit, so it's not possible to create two different instances with the same parameters. The first two assignments are manipulating the same global while the third one is manipulating a different instance. In the Ada case, the three instances are explicitly created, named, and referenced individually.

53.3 Generic Parameters

Ada offers a wide variety of generic parameters which is difficult to translate into other languages. The parameters used during instantiation — and as a consequence those on which the generic unit may rely on — may be variables, types, or subprograms with certain properties. For example, the following provides a sort algorithm for any kind of array:

```

generic
  type Component is private;
  type Index is (<);
  with function "<" (Left, Right : Component) return Boolean;
  type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);

```

The above declaration states that we need a type (Component), a discrete type (Index), a comparison subprogram ("<"), and an array definition (Array_Type). Given these, it's possible to write an algorithm that can sort any Array_Type. Note the usage of the **with** reserved word in front of the function name, to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```

type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type, such as String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type, T2 is its designated type

```


EXCEPTIONS

Exceptions are a mechanism for dealing with run-time occurrences that are rare, that usually correspond to errors (such as improperly formed input data), and whose occurrence causes an unconditional transfer of control.

54.1 Standard Exceptions

Compared with Java and C++, the notion of an Ada exception is very simple. An exception in Ada is an object whose "type" is **exception**, as opposed to classes in Java or any type in C++. The only piece of user data that can be associated with an Ada exception is a String. Basically, an exception in Ada can be raised, and it can be handled; information associated with an occurrence of an exception can be interrogated by a handler.

Ada makes heavy use of exceptions especially for data consistency check failures at run time. These include, but are not limited to, checking against type ranges and array boundaries, null pointers, various kind of concurrency properties, and functions not returning a value. For example, the following piece of code will raise the exception `Constraint_Error`:

```
procedure P is
  V : Positive;
begin
  V := -1;
end P;
```

In the above code, we're trying to assign a negative value to a variable that's declared to be positive. The range check takes place during the assignment operation, and the failure raises the `Constraint_Error` exception at that point. (Note that the compiler may give a warning that the value is out of range, but the error is manifest as a run-time exception.) Since there is no local handler, the exception is propagated to the caller; if P is the main procedure, then the program will be terminated.

Java and C++ can **throw** and **catch** exceptions when **trying** code. All Ada code is already implicitly within **try** blocks, and exceptions are raised and handled.

[Ada]

```
begin
  Some_Call;
exception
  when Exception_1 =>
    Put_Line ("Error 1");
  when Exception_2 =>
    Put_Line ("Error 2");
  when others =>
    Put_Line ("Unknown error");
end;
```

[C++]


```
try {
    someCall ();
} catch (Exception1) {
    cout << "Error 1" << endl;
} catch (Exception2) {
    cout << "Error 2" << endl;
} catch (...) {
    cout << "Unknown error" << endl;
}
```

[Java]

```
try {
    someCall ();
} catch (Exception1 e1) {
    System.out.println ("Error 1");
} catch (Exception2 e2) {
    System.out.println ("Error 2");
} catch (Throwable e3) {
    System.out.println ("Unknown error");
}
```

Raising and throwing exceptions is permissible in all three languages.

54.2 Custom Exceptions

Custom exception declarations resemble object declarations, and they can be created in Ada using the **exception** keyword:

```
My_Exception : exception;
```

Your exceptions can then be raised using a **raise** statement, optionally accompanied by a message following the **with** reserved word:

[Ada]

```
raise My_Exception with "Some message";
```

[C++]

```
throw My_Exception ("Some message");
```

[Java]

```
throw new My_Exception ("Some message");
```

Language defined exceptions can also be raised in the same manner:

```
raise Constraint_Error;
```

CONCURRENCY

55.1 Tasks

Java and Ada both provide support for concurrency in the language. The C++ language has added a concurrency facility in its most recent revision, C++11, but we are assuming that most C++ programmers are not (yet) familiar with these new features. We thus provide the following mock API for C++ which is similar to the Java Thread class:

```
class Thread {
public:
    virtual void run (); // code to execute
    void start (); // starts a thread and then call run ()
    void join (); // waits until the thread is finished
};
```

Each of the following examples will display the 26 letters of the alphabet twice, using two concurrent threads/tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

```
procedure Main is -- implicitly called by the environment task
    task My_Task;

    task body My_Task is
    begin
        for I in 'A' .. 'Z' loop
            Put_Line (I);
        end loop;
    end My_Task;
begin
    for I in 'A' .. 'Z' loop
        Put_Line (I);
    end loop;
end Main;
```

[C++]

```
class MyThread : public Thread {
public:
    void run () {
        for (char i = 'A'; i <= 'Z'; ++i) {
            cout << i << endl;
        }
    }
};
```

(continues on next page)

(continued from previous page)

```

int main (int argc, char ** argv) {
    MyThread myTask;
    myTask.start ();

    for (char i = 'A'; i <= 'Z'; ++i) {
        cout << i << endl;
    }

    myTask.join ();

    return 0;
}

```

[Java]

```

public class Main {
    static class MyThread extends Thread {
        public void run () {
            for (char i = 'A'; i <= 'Z'; ++i) {
                System.out.println (i);
            }
        }
    }

    public static void main (String args) {
        MyThread myTask = new MyThread ();
        myTask.start ();

        for (char i = 'A'; i <= 'Z'; ++i) {
            System.out.println (i);
        }
        myTask.join ();
    }
}

```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the **begin**. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

An Ada task type is somewhat analogous to a Java Thread subclass, but in Java the instances of such a subclass are always dynamically allocated. In Ada an instance of a task type may either be declared or dynamically allocated.

Task types can be parametrized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the implementation's task scheduling algorithm.

[Ada]

```

task type My_Task (First : Character);

task body My_Task is
begin
    for I in First .. 'Z' loop

```

(continues on next page)

(continued from previous page)

```

        Put_Line (I);
    end loop;
end My_Task;

procedure Main is
    Tab : array (0 .. 9) of My_Task ('G');
begin
    null;
end Main;

```

[C++]

```

class MyThread : public Thread {
public:

    char first;

    void run () {
        for (char i = first; i <= 'Z'; ++i) {
            cout << i << endl;
        }
    }
};

int main (int argc, char ** argv) {
    MyThread tab [10];

    for (int i = 0; i < 9; ++i) {
        tab [i].first = 'G';
        tab [i].start ();
    }

    for (int i = 0; i < 9; ++i) {
        tab [i].join ();
    }

    return 0;
}

```

[Java]

```

public class MyThread extends Thread {
    public char first;

    public MyThread (char first){
        this.first = first;
    }

    public void run () {
        for (char i = first; i <= 'Z'; ++i) {
            cout << i << endl;
        }
    }
}

public class Main {
    public static void main (String args) {
        MyThread [] tab = new MyThread [10];

        for (int i = 0; i < 9; ++i) {
            tab [i] = new MyThread ('G');
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        tab [i].start ();
    }

    for (int i = 0; i < 9; ++i) {
        tab [i].join ();
    }
}

```

In Ada a task may be allocated on the heap as opposed to the stack. The task will then start as soon as it has been allocated, and terminates when its work is completed. This model is probably the one that's the most similar to Java:

[Ada]

```

type Ptr_Task is access My_Task;

procedure Main is
    T : Ptr_Task;
begin
    T := new My_Task ('G');
end Main;

```

[C++]

```

int main (int argc, char ** argv) {
    MyThread * t = new MyThread ();
    t->first = 'G';
    t->start ();
    return 0;
}

```

[Java]

```

public class Main {
    public static void main (String args) {
        MyThread t = new MyThread ('G');

        t.start ();
    }
}

```

55.2 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Ada's rendezvous facility cannot be modeled with C++ or Java without complex machinery. Therefore, this section will just show examples written in Ada.

Let's consider the following example:

```

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    task After is
        entry Go;
    end After ;

    task body After is

```

(continues on next page)

(continued from previous page)

```

begin
  accept Go;
  Put_Line ("After");
end After;

begin
  Put_Line ("Before");
  After.Go;
end;

```

The Go **entry** declared in After is the external interface to the task. In the task body, the **accept** statement causes the task to wait for a call on the entry. This particular **entry** and **accept** pair doesn't do much more than cause the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The **entry/accept** pair can take/pass parameters, and the **accept** statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

```

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  task After is
    entry Go (Text : String);
  end After ;

  task body After is
  begin
    accept Go (Text : String) do
      Put_Line ("After: " & Text);
    end Go;
  end After;

begin
  Put_Line ("Before");
  After.Go ("Main");
end;

```

In the above example, the Put_Line is placed in the **accept** statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the **begin** of Main, task After is started and the main procedure is suspended.
2. After reaches the **accept** statement and is suspended, since there is no pending call on the Go entry.
3. The main procedure is awakened and executes the Put_Line invocation, displaying the string "Before".
4. The main procedure calls the Go entry. Since After is suspended on its **accept** statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task After is awakened to execute the body of the **accept** statement. The actual parameter "Main" is passed to the **accept** statement, and the Put_Line invocation is executed. As a result, the string "After: Main" is displayed.
6. When the **accept** statement is completed, both the After task and the main procedure are ready to run. Suppose that the Main procedure is given the processor. It

reaches its **end**, but the local task After has not yet terminated. The main procedure is suspended.

7. The After task continues, and terminates since it is at its **end**. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

55.3 Selective Rendezvous

The accept statement by itself can only wait for a single event (call) at a time. The **select** statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call Increment, Decrement, and Get:

```
task Counter is
  entry Get (Result : out Integer);
  entry Increment;
  entry Decrement;
end Counter;

task body Counter is
  Value : Integer := 0;
begin
  loop
    select
      accept Increment do
        Value := Value + 1;
      end Increment;
    or
      accept Decrement do
        Value := Value - 1;
      end Decrement;
    or
      accept Get (Result : out Integer) do
        Result := Value;
      end Get;
    or
      delay 60.0; -- delay 1 minute
      exit;
    end select;
  end loop;
end Counter;
```

When the task's statement flow reaches the **select**, it will wait for all four events — three entries and a delay — in parallel. If the delay of one minute is exceeded, the task will execute the statements following the **delay** statement (and in this case will exit the loop, in effect terminating the task). The accept bodies for the Increment, Decrement, or Get entries will be otherwise executed as they're called. These four sections of the **select** statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a "race condition" could occur where multiple tasks simultaneously calling, say, Increment, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call Increment then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended at the **select** statement) then the calling task is suspended, and placed in the queue of

the entry that it is calling. From the perspective of the Counter task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the one minute delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the **select** branch with an **accept** statement for that entry. The choice of which caller to accept, if more than one, depends on the queuing policy, which can be specified via a pragma defined in the Real-Time Systems Annex of the Ada standard; the default is First-In First-Out.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued (the choices depend on the queueing policy).

55.4 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a *protected object*, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its *protected operations*) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

```
protected Counter is
  function Get return Integer;
  procedure Increment;
  procedure Decrement;
private
  Value : Integer := 0;
end Counter;

protected body Counter is
  function Get return Integer is
  begin
    return Value;
  end Get;

  procedure Increment is
  begin
    Value := Value + 1;
  end Increment;

  procedure Decrement is
  begin
    Value := Value - 1;
  end Decrement;
end Counter;
```

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between

an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

```
Counter.Increment;  
Counter.Decrement;  
Put_Line (Integer'Image (Counter.Get));
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare  
  protected type Counter is  
    -- as above  
  end Counter;  
  
  protected body Counter is  
    -- as above  
  end Counter;  
  
  C1 : Counter;  
  C2 : Counter;  
begin  
  C1.Increment;  
  C2.Decrement;  
  ...  
end;
```

Protected objects and types can declare a procedure-like operation known as an "entry". An entry is somewhat similar to a procedure but includes a so-called *barrier condition* that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

```
protected type Binary_Semaphore is  
  entry Wait;  
  procedure Signal;  
private  
  Signaled : Boolean := False;  
end Binary_Semaphore;  
  
protected body Binary_Semaphore is  
  entry Wait when Signaled is  
  begin  
    Signaled := False;  
  end Wait;  
  
  procedure Signal is  
  begin  
    Signaled := True;
```

(continues on next page)

(continued from previous page)

```
end Signal;  
end Binary_Semaphore;
```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

LOW LEVEL PROGRAMMING

56.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
  V  : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record
with Pack;
```

[C++]

```
struct R {
  unsigned int v:8;
  bool b1;
  bool b2;
};
```

[Java]

```
public class R {
  public byte v;
  public boolean b1;
  public boolean b2;
}
```

The Ada and the C++ code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C++ and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C++, the number of bits required by each field needs to be specified. Here, we're stating that *v* is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The *Pack* aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

```
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record;

for R use record
  -- Occupy the first bit of the first byte.
  B1 at 0 range 0 .. 0;

  -- Occupy the last 7 bits of the first byte,
  -- as well as the first bit of the second byte.
  V at 0 range 1 .. 8;

  -- Occupy the second bit of the second byte.
  B2 at 1 range 1 .. 1;
end record;
```

We omit the **with** Pack directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type R across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

56.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the `rdtsc` instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64 bit value split between the `edx` and `eax` registers.

GNAT provides a subprogram called `System.Machine_Code.Asm` that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

```
with System.Machine_Code; use System.Machine_Code;
with Interfaces;          use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
  Low, High : Unsigned_32;
  Counter   : Unsigned_64;
begin
  Asm ("rdtsc",
      Outputs =>
        (Unsigned_32'Asm_Output ("=a", Low),
         Unsigned_32'Asm_Output ("=d", High)),
      Volatile => True);
```

(continues on next page)

(continued from previous page)

```

Counter :=
  Unsigned_64 (High) * 2 ** 32 +
  Unsigned_64 (Low);

return Counter;
end Get_Processor_Cycles;

```

The Unsigned_32 'Asm_Output' clauses above provide associations between machine registers and source-level variables to be updated. "`=a`" and "`=d`" refer to the `eax` and `edx` machine registers, respectively. The use of the Unsigned_32 and Unsigned_64 types from package Interfaces ensures correct representation of the data. We assemble the two 32-bit values to form a single 64 bit value.

We set the Volatile parameter to `True` to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the `eax` and `edx` registers to implement the `High` and `Low` variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

56.3 Interfacing with C

Much effort was spent making Ada easy to interface with other languages. The Interfaces package hierarchy and the pragmas `Convention`, `Import`, and `Export` allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

```

struct my_struct {
    int A, B;
};

void call (my_struct * p) {
    printf ("%d", p->A);
}

```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C `struct my_struct` is represented, we can use the following record along with a `pragma Convention`. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

```

type my_struct is record
  A : Interfaces.C.int;
  B : Interfaces.C.int;
end record;
pragma Convention (C, my_struct);

```

Describing a foreign subprogram call to Ada code is called "binding" and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a `void` function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a `pragma Import`:

```
procedure Call (V : my_struct);  
pragma Import (C, Call, "call"); -- Third argument optional
```

The Import pragma specifies that whenever Call is invoked by Ada code, it should invoke the call function with the C calling convention.

And that's all that's necessary. Here's an example of a call to Call:

```
declare  
  V : my_struct := (A => 1, B => 2);  
begin  
  Call (V);  
end;
```

You can also make Ada subprograms available to C code, and examples of this can be found in the GNAT User's Guide. Interfacing with C++ and Java use implementation-dependent features that are also available with GNAT.

CONCLUSION

All the usual paradigms of imperative programming can be found in all three languages that we surveyed in this document. However, Ada is different from the rest in that it's more explicit when expressing properties and expectations. This is a good thing: being more formal affords better communication among programmers on a team and between programmers and machines. You also get more assurance of the coherence of a program at many levels. Ada can help reduce the cost of software maintenance by shifting the effort to creating a sound system the first time, rather than working harder, more often, and at greater expense, to fix bugs found later in systems already in production. Applications that have reliability needs, long term maintenance requirements, or safety/security concerns are those for which Ada has a proven track record.

It's becoming increasingly common to find systems implemented in multiple languages, and Ada has standard interfacing facilities to allow Ada code to invoke subprograms and/or reference data structures from other language environments, or vice versa. Use of Ada thus allows easy interfacing between different technologies, using each for what it's best at.

We hope this guide has provided some insight into the Ada software engineer's world and has made Ada more accessible to programmers already familiar with programming in other languages.

REFERENCES

The Ada Information Clearinghouse website <http://www.adaic.org/learn/materials/>, maintained by the Ada Resource Association, contains links to a variety of training materials (books, articles, etc.) that can help in learning Ada. The Development Center page <http://www.adacore.com/knowledge> on AdaCore's website also contains links to useful information including vides and tutorials on Ada.

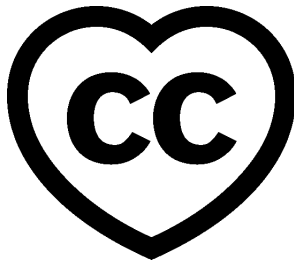
The most comprehensive textbook is John Barnes' *Programming in Ada 2012*, which is oriented towards professional software developers.

Part V

Introduction To SPARK

Copyright © 2018 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)³⁸⁵



This tutorial is an interactive introduction to the SPARK programming language and its formal verification tools. You will learn the difference between Ada and SPARK and how to use the various analysis tools that come with SPARK.

This document was prepared by Claire Dross and Yannick Moy.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](https://learn.adacore.com/zip/learning-ada_code.zip)³⁸⁶. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

³⁸⁵ [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

³⁸⁶ https://learn.adacore.com/zip/learning-ada_code.zip

SPARK OVERVIEW

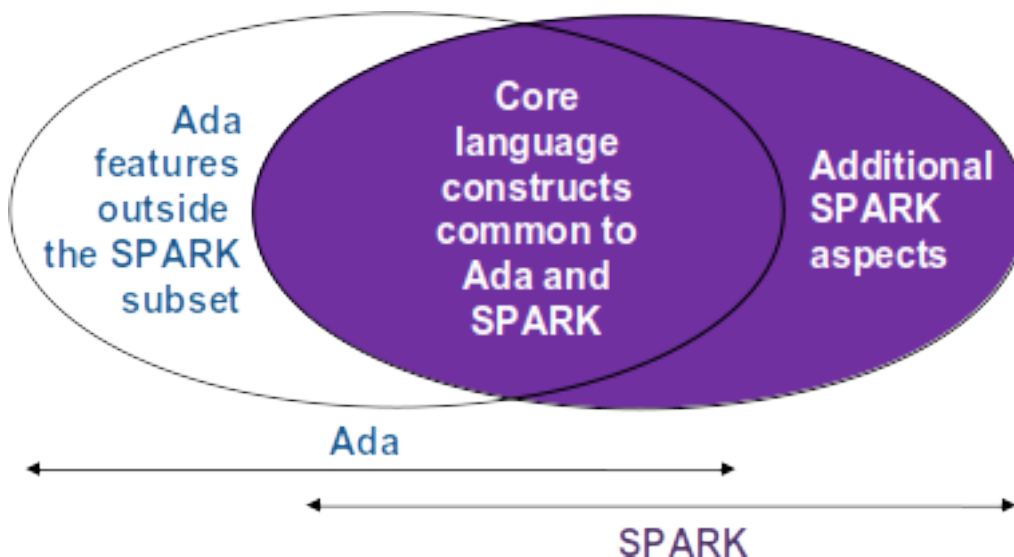
This tutorial is an introduction to the SPARK programming language and its formal verification tools. You need not know any specific programming language (although going over the [Introduction to Ada course](#) (page 5) first may help) or have experience in formal verification.

59.1 What is it?

SPARK refers to two different things:

- a programming language targeted at functional specification and static verification, and
- a set of development and verification tools for that language.

The SPARK language is based on a subset of the Ada language. Ada is particularly well suited to formal verification since it was designed for critical software development. SPARK builds on that foundation.



Version 2012 of Ada introduced the use of *aspects*, which can be used for subprogram contracts, and version 2014 of SPARK added its own aspects to further aid static analysis.

59.2 What do the tools do?

We start by reviewing static verification of programs, which is verification of the source code performed without compiling or executing it. Verification uses tools that perform static analysis. These can take various forms. They include tools that check types and enforce visibility rules, such as the compiler, in addition to those that perform more complex reasoning,

such as abstract interpretation, as done by a tool like the [GNAT Static Analysis Suite](#)³⁸⁷ from AdaCore. The tools that come with SPARK perform two different forms of static analysis:

- *flow analysis* is the fastest form of analysis. It checks initializations of variables and looks at data dependencies between inputs and outputs of subprograms. It can also find unused assignments and unmodified variables.
- *proof* checks for the absence of runtime errors as well as the conformance of the program with its specifications.

59.3 Key Tools

The tool for formal verification of the SPARK language is called *GNATprove*. It checks for conformance with the SPARK subset and performs flow analysis and proof of the source code. Several other tools support the SPARK language, including both the [GNAT compiler](#)³⁸⁸ and the [GNAT Studio integrated development environment](#)³⁸⁹.

59.4 A trivial example

We start with a simple example of a subprogram in Ada that uses SPARK aspects to specify verifiable subprogram contracts. The subprogram, called *Increment*, adds 1 to the value of its parameter *X*:

Listing 1763: increment.ads

```
1 procedure Increment
2   (X : in out Integer)
3 with
4   Global    => null,
5   Depends  => (X => X),
6   Pre       => X < Integer'Last,
7   Post      => X = X'Old + 1;
```

Listing 1764: increment.adb

```
1 procedure Increment
2   (X : in out Integer)
3 is
4 begin
5   X := X + 1;
6 end Increment;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Trivial_Example
MD5: ce28b1facb44917b6cc208639c187064

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
increment.adb:5:10: info: overflow check proved
increment.ads:4:03: info: data dependencies proved
increment.ads:5:03: info: flow dependencies proved
```

(continues on next page)

³⁸⁷ <https://www.adacore.com/static-analysis-suite>

³⁸⁸ <https://www.adacore.com/gnatpro>

³⁸⁹ <https://www.adacore.com/gnatpro/toolsuite/gps>

(continued from previous page)

```
increment.ads:7:14: info: postcondition proved
increment.ads:7:24: info: overflow check proved
```

The contracts are written using the Ada *aspect* feature and those shown specify several properties of this subprogram:

- The SPARK Global aspect says that Increment does not read or write any global variables.
- The SPARK Depend aspect is especially interesting for security: it says that the value of the parameter *X* after the call depends only on the (previous) value of *X*.
- The Pre and Post aspects of Ada specify functional properties of Increment:
 - Increment is only allowed to be called if the value of *X* prior to the call is less than **Integer'Last**. This ensures that the addition operation performed in the subprogram body doesn't overflow.
 - Increment does indeed perform an increment of *X*: the value of *X* after a call is one greater than its value before the call.

GNATprove can verify all of these contracts. In addition, it verifies that no error can be raised at runtime when executing Increment's body.

59.5 The Programming Language

It's important to understand why there are differences between the SPARK and Ada languages. The aim when designing the SPARK subset of Ada was to create the largest possible subset of Ada that was still amenable to simple specification and sound verification.

The most notable restrictions from Ada are related to exceptions and access types, both of which are known to considerably increase the amount of user-written annotations required for full support. Backwards goto statements and controlled types are also not supported since they introduce non-trivial control flow. The two remaining restrictions relate to side-effects in expressions and aliasing of names, which we now cover in more detail.

59.6 Limitations

59.6.1 No side-effects in expressions

The SPARK language doesn't allow side-effects in expressions. In other words, evaluating a SPARK expression must not update any object. This limitation is necessary to avoid unpredictable behavior that depends on order of evaluation, parameter passing mechanisms, or compiler optimizations. The expression for *Dummy* below is non-deterministic due to the order in which the two calls to *F* are evaluated. It's therefore not legal SPARK.

Listing 1765: show_illegal_ada_code.adb

```

1  procedure Show_Illegal_Ada_Code is
2
3      function F (X : in out Integer) return Integer is
4          Tmp : constant Integer := X;
5      begin
6          X := X + 1;
7          return Tmp;
8      end F;
9
10     Dummy : Integer := 0;
```

(continues on next page)

(continued from previous page)

```
11
12 begin
13   Dummy := F (Dummy) - F (Dummy); -- ??
14 end Show_Illegal_Ada_Code;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Illegal_Ada_Code
MD5: a5cbf1824526857da94791ac1790200c

Build output

```
show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
↳ because order of evaluation is arbitrary
gprbuild: *** compilation phase failed
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
↳ because order of evaluation is arbitrary
gnatprove: error during generation of Global contracts
```

In fact, the code above is not even legal Ada, so the same error is generated by the GNAT compiler. But SPARK goes further and GNATprove also produces an error for the following equivalent code that is accepted by the Ada compiler:

Listing 1766: show_illegal_spark_code.adb

```
1 procedure Show_Illegal_SPARK_Code is
2
3   Dummy : Integer := 0;
4
5   function F return Integer is
6     Tmp : constant Integer := Dummy;
7   begin
8     Dummy := Dummy + 1;
9     return Tmp;
10  end F;
11
12 begin
13   Dummy := F - F; -- ??
14 end Show_Illegal_SPARK_Code;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Illegal_SPARK_Code
MD5: e747edb6ee147adb7fba97c9e7c8d5ef

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_illegal_spark_code.adb:5:13: error: function "F" with output global "Dummy"
↳ is not allowed in SPARK [E0005]
show_illegal_spark_code.adb:5:13: error: violation of pragma SPARK_Mode at /
↳ vagrant/frontend/dist/test_output/projects/Courses/Intro_To_Spark/Overview/
↳ Illegal_SPARK_Code/e747edb6ee147adb7fba97c9e7c8d5ef/main_spark.adc:12
show_illegal_spark_code.adb:5:13: error: launch "gnatprove --explain=E0005" for
```

(continues on next page)

(continued from previous page)

→more information
 gnatprove: error during analysis of data and information flow

The SPARK languages enforces the lack of side-effects in expressions by forbidding side-effects in functions, which include modifications to either parameters or global variables. As a consequence, SPARK forbids functions with **out** or **in out** parameters in addition to functions modifying a global variable. Function F below is illegal in SPARK, while Function Incr might be legal if it doesn't modify any global variables and function Incr_And_Log might be illegal if it modifies global variables to perform logging.

```
function F (X : in out Integer) return Integer;    -- Illegal
function Incr (X : Integer) return Integer;        -- OK?
function Incr_And_Log (X : Integer) return Integer; -- OK?
```

In most cases, you can easily replace these functions by procedures with an **out** parameter that returns the computed value.

When it has access to function bodies, GNATprove verifies that those functions are indeed free from side-effects. Here for example, the two functions Incr and Incr_And_Log have the same signature, but only Incr is legal in SPARK. Incr_And_Log isn't: it attempts to update the global variable Call_Count.

Listing 1767: side_effects.ads

```
1 package Side_Effects is
2
3   function Incr (X : Integer) return Integer;    -- OK?
4
5   function Incr_And_Log (X : Integer) return Integer; -- OK?
6
7 end Side_Effects;
```

Listing 1768: side_effects.adb

```
1 package body Side_Effects is
2
3   function Incr (X : Integer) return Integer
4   is (X + 1); -- OK
5
6   Call_Count : Natural := 0;
7
8   function Incr_And_Log (X : Integer) return Integer is
9   begin
10    Call_Count := Call_Count + 1; -- Illegal
11    return X + 1;
12  end Incr_And_Log;
13
14 end Side_Effects;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Side_Effects
 MD5: 1b555e4b7bb519eea4df718a9356a2ed

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...

(continues on next page)

(continued from previous page)

```

Phase 3 of 3: analysis of data and information flow ...
side_effects.ads:5:13: error: function "Incr_And_Log" with output global "Call_
↳Count" is not allowed in SPARK [E0005]
side_effects.ads:5:13: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳dist/test_output/projects/Courses/Intro_To_Spark/Overview/Side_Effects/
↳1b555e4b7bb519eea4df718a9356a2ed/main_spark.adc:12
side_effects.ads:5:13: error: launch "gnatprove --explain=E0005" for more_
↳information
gnatprove: error during analysis of data and information flow

```

59.6.2 No aliasing of names

Another restriction imposed by the SPARK subset concerns [aliasing](#)³⁹⁰. We say that two names are *aliased* if they refer to the same object. There are two reasons why aliasing is forbidden in SPARK:

- It makes verification more difficult because it requires taking into account the fact that modifications to variables with different names may actually update the same object.
- Results may seem unexpected from a user point of view. The results of a subprogram call may depend on compiler-specific attributes, such as parameter passing mechanisms, when its parameters are aliased.

Aliasing can occur as part of the parameter passing that occurs in a subprogram call. Functions have no side-effects in SPARK, so aliasing of parameters in function calls isn't problematic; we need only consider procedure calls. When a procedure is called, SPARK verifies that no **out** or **in out** parameter is aliased with either another parameter of the procedure or a global variable modified in the procedure's body.

Procedure `Move_To_Total` is an example where the possibility of aliasing wasn't taken into account by the programmer:

Listing 1769: no_aliasing.adb

```

1  procedure No_Aliasing is
2
3      Total : Natural := 0;
4
5      procedure Move_To_Total (Source : in out Natural)
6          with Post => Total = Total'Old + Source'Old and Source = 0
7      is
8          begin
9              Total := Total + Source;
10             Source := 0;
11         end Move_To_Total;
12
13         X : Natural := 3;
14
15     begin
16         Move_To_Total (X);           -- OK
17         pragma Assert (Total = 3);   -- OK
18         Move_To_Total (Total);       -- flow analysis error
19         pragma Assert (Total = 6);   -- runtime error
20     end No_Aliasing;

```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Aliasing
MD5: 91038ef030fe27e3b000ab3db9c134ad

³⁹⁰ [https://en.wikipedia.org/wiki/Aliasing_\(computing\)](https://en.wikipedia.org/wiki/Aliasing_(computing))

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
no_aliasing.adb:18:19: high: formal parameter "Source" and global "Total" are
↳ aliased (SPARK RM 6.4.2)
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : no_aliasing.adb:19
```

`Move_To_Total` adds the value of its input parameter `Source` to the global variable `Total` and then resets `Source` to 0. The programmer has clearly not taken into account the possibility of an aliasing between `Total` and `Source`. (This sort of error is quite common.)

This procedure itself is valid SPARK. When doing verification, GNATprove assumes, like the programmer did, that there's no aliasing between `Total` and `Source`. To ensure this assumption is valid, GNATprove checks for possible aliasing on every call to `Move_To_Total`. Its final call in procedure `No_Aliasing` violates this assumption, which produces both a message from GNATprove and a runtime error (an assertion violation corresponding to the expected change in `Total` from calling `Move_To_Total`). Note that the postcondition of `Move_To_Total` is not violated on this second call since integer parameters are passed by copy and the postcondition is checked before the copy-back from the formal parameters to the actual arguments.

Aliasing can also occur as a result of using access types ([pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming))³⁹¹ in Ada). These are restricted in SPARK so that only benign aliasing is allowed, when both names are only used to read the data. In particular, assignment between access objects operates a transfer of ownership, where the source object loses its permission to read or write the underlying allocated memory.

Procedure `Ownership_Transfer` is an example of code that is legal in Ada but rejected in SPARK due to aliasing:

Listing 1770: `ownership_transfer.adb`

```
1  procedure Ownership_Transfer is
2      type Int_Ptr is access Integer;
3      X      : Int_Ptr;
4      Y      : Int_Ptr;
5      Dummy  : Integer;
6  begin
7      X      := new Integer'(1);
8      X.all := X.all + 1;
9      Y      := X;
10     Y.all := Y.all + 1;
11     X.all := X.all + 1;  -- illegal
12     X.all := 1;         -- illegal
13     Dummy := X.all;     -- illegal
14 end Ownership_Transfer;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.Overview.Ownership_Transfer
MD5: 951fe1c930d43a5009e607994ae0dd03
```

Prover output

³⁹¹ [https://en.wikipedia.org/wiki/Pointer_\(computer_programming\)](https://en.wikipedia.org/wiki/Pointer_(computer_programming))

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
ownership_transfer.adb:11:06: error: dereference from "X" is not writable
ownership_transfer.adb:11:06: error: object was moved at line 9 [E0010]
ownership_transfer.adb:11:06: error: launch "gnatprove --explain=E0010" for more
↳ information
ownership_transfer.adb:11:15: error: dereference from "X" is not readable
ownership_transfer.adb:11:15: error: object was moved at line 9 [E0010]
ownership_transfer.adb:11:15: error: launch "gnatprove --explain=E0010" for more
↳ information
ownership_transfer.adb:12:06: error: dereference from "X" is not writable
ownership_transfer.adb:12:06: error: object was moved at line 9 [E0010]
ownership_transfer.adb:12:06: error: launch "gnatprove --explain=E0010" for more
↳ information
ownership_transfer.adb:13:15: error: dereference from "X" is not readable
ownership_transfer.adb:13:15: error: object was moved at line 9 [E0010]
ownership_transfer.adb:13:15: error: launch "gnatprove --explain=E0010" for more
↳ information
gnatprove: error during analysis of data and information flow
```

After the assignment of X to Y, variable X cannot be used anymore to read or write the underlying allocated memory.

Note

For more details on these limitations, see the [SPARK User's Guide](#)³⁹².

59.7 Designating SPARK Code

Since the SPARK language is restricted to only allow easily specifiable and verifiable constructs, there are times when you can't or don't want to abide by these limitations over your entire code base. Therefore, the SPARK tools only check conformance to the SPARK subset on code which you identify as being in SPARK.

You do this by using an aspect named `SPARK_Mode`. If you don't explicitly specify otherwise, `SPARK_Mode` is *Off*, meaning you can use the complete set of Ada features in that code and that it should not be analyzed by GNATprove. You can change this default either selectively (on some units or subprograms or packages inside units) or globally (using a configuration pragma, which is what we're doing in this tutorial). To allow simple reuse of existing Ada libraries, entities declared in imported units with no explicit `SPARK_Mode` can still be used from SPARK code. The tool only checks for SPARK conformance on the declaration of those entities which are actually used within the SPARK code.

Here's a common case of using the `SPARK_Mode` aspect:

```
package P
  with SPARK_Mode => On
is
  -- package spec is IN SPARK, so can be used by SPARK clients
end P;

package body P
  with SPARK_Mode => Off
is
```

(continues on next page)

³⁹² https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#language-restrictions

(continued from previous page)

```
-- body is NOT IN SPARK, so is ignored by GNATprove
end P;
```

The package P only defines entities whose specifications are in the SPARK subset. However, it wants to use all Ada features in its body. Therefore the body should not be analyzed and has its SPARK_Mode aspect set to *Off*.

You can specify SPARK_Mode in a fine-grained manner on a per-unit basis. An Ada package has four different components: the visible and private parts of its specification and the declarative and statement parts of its body. You can specify SPARK_Mode as being either *On* or *Off* on any of those parts. Likewise, a subprogram has two parts: its specification and its body.

A general rule in SPARK is that once SPARK_Mode has been set to *Off*, it can never be switched *On* again in the same part of a package or subprogram. This prevents setting SPARK_Mode to *On* for local units of a unit with SPARK_Mode *Off* and switching back to SPARK_Mode *On* for a part of a given unit where it was set to *Off* in a previous part.

Note

For more details on the use of SPARK_Mode, see the [SPARK User's Guide](#)³⁹³.

59.8 Code Examples / Pitfalls

59.8.1 Example #1

Here's a package defining an abstract stack type (defined as a private type in SPARK) of Element objects along with some subprograms providing the usual functionalities of stacks. It's marked as being in the SPARK subset.

Listing 1771: stack_package.ads

```
1 package Stack_Package
2   with SPARK_Mode => On
3 is
4   type Element is new Natural;
5   type Stack is private;
6
7   function Empty return Stack;
8   procedure Push (S : in out Stack; E : Element);
9   function Pop (S : in out Stack) return Element;
10
11 private
12   type Stack is record
13     Top : Integer;
14     -- ...
15   end record;
16
17 end Stack_Package;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_01
MD5: 2b15e13e850435fb93406054d70b51c6

Prover output

³⁹³ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/spark_mode.html


```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
stack_package.ads:9:18: error: function cannot have parameter of mode "out" or "in_
↳out" in SPARK [E0015]
stack_package.ads:9:18: error: launch "gnatprove --explain=E0015" for more_
↳information
gnatprove: error during generation of Global contracts
```

Side-effects in expressions are not allowed in SPARK. Therefore, Pop is not allowed to modify its parameter S.

59.8.2 Example #2

Let's turn to an abstract state machine version of a stack, where the unit provides a single instance of a stack. The content of the stack (global variables Content and Top) is not directly visible to clients. In this stripped-down version, only the function Pop is available to clients. The package spec and body are marked as being in the SPARK subset.

Listing 1772: global_stack.ads

```
1 package Global_Stack
2   with SPARK_Mode => On
3 is
4   type Element is new Integer;
5
6   function Pop return Element;
7
8 end Global_Stack;
```

Listing 1773: global_stack.adb

```
1 package body Global_Stack
2   with SPARK_Mode => On
3 is
4   Max : constant Natural := 100;
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array;
8   Top     : Natural;
9
10  function Pop return Element is
11    E : constant Element := Content (Top);
12  begin
13    Top := Top - 1;
14    return E;
15  end Pop;
16
17 end Global_Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_02
MD5: 8c4eb564643eef48264b5e43a6f580b9

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
global_stack.ads:6:13: error: function "Pop" with output global "Top" is not_
(continues on next page)
```

(continued from previous page)

```

↳allowed in SPARK [E0005]
global_stack.ads:6:13: error: violation of aspect SPARK_Mode at line 2
global_stack.ads:6:13: error: launch "gnatprove --explain=E0005" for more
↳information
gnatprove: error during analysis of data and information flow

```

As above, functions should be free from side-effects. Here, Pop updates the global variable Top, which is not allowed in SPARK.

59.8.3 Example #3

We now consider two procedures: Permute and Swap. Permute applies a circular permutation to the value of its three parameters. Swap then uses Permute to swap the value of X and Y.

Listing 1774: p.ads

```

1 package P
2   with SPARK_Mode => On
3 is
4   procedure Permute (X, Y, Z : in out Positive);
5   procedure Swap (X, Y : in out Positive);
6 end P;

```

Listing 1775: p.adb

```

1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Permute (X, Y, Z : in out Positive) is
5     Tmp : constant Positive := X;
6   begin
7     X := Y;
8     Y := Z;
9     Z := Tmp;
10  end Permute;
11
12  procedure Swap (X, Y : in out Positive) is
13  begin
14    Permute (X, Y, Y);
15  end Swap;
16 end P;

```

Listing 1776: test_swap.adb

```

1 with P; use P;
2
3 procedure Test_Swap
4   with SPARK_Mode => On
5 is
6   A : Integer := 1;
7   B : Integer := 2;
8 begin
9   Swap (A, B);
10 end Test_Swap;

```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_03
MD5: 0868a806061d86af4d2a03b1e7dc83c2

Build output

```
p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"  
gprbuild: *** compilation phase failed
```

Prover output

```
Phase 1 of 3: generation of data representation information ...  
Phase 2 of 3: generation of Global contracts ...  
p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"  
gnatprove: error during generation of Global contracts
```

Here, the values for parameters Y and Z are aliased in the call to `Permute`, which is not allowed in SPARK. In fact, in this particular case, this is even a violation of Ada rules so the same error is issued by the Ada compiler.

In this example, we see the reason why aliasing is not allowed in SPARK: since Y and Z are **Positive**, they are passed by copy and the result of the call to `Permute` depends on the order in which they're copied back after the call.

59.8.4 Example #4

Here, the `Swap` procedure is used to swap the value of the two record components of R.

Listing 1777: p.ads

```
1 package P  
2   with SPARK_Mode => On  
3 is  
4   type Rec is record  
5     F1 : Positive;  
6     F2 : Positive;  
7   end record;  
8  
9   procedure Swap_Fields (R : in out Rec);  
10  procedure Swap (X, Y : in out Positive);  
11 end P;
```

Listing 1778: p.adb

```
1 package body P  
2   with SPARK_Mode => On  
3 is  
4   procedure Swap (X, Y : in out Positive) is  
5     Tmp : constant Positive := X;  
6   begin  
7     X := Y;  
8     Y := Tmp;  
9   end Swap;  
10  
11  procedure Swap_Fields (R : in out Rec) is  
12  begin  
13    Swap (R.F1, R.F2);  
14  end Swap_Fields;  
15  
16 end P;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.Overview.Example_04  
MD5: ae4d3ebe8dd1a8f67f35cedffdea2ac9
```

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...

This code is correct. The call to Swap is safe: two different components of the same record can't refer to the same object.

59.8.5 Example #5

Here's a slight modification of the previous example using an array instead of a record: Swap_Indexes calls Swap on values stored in the array A.

Listing 1779: p.ads

```

1 package P
2   with SPARK_Mode => On
3 is
4   type P_Array is array (Natural range <>) of Positive;
5
6   procedure Swap_Indexes (A : in out P_Array; I, J : Natural);
7   procedure Swap (X, Y : in out Positive);
8 end P;
```

Listing 1780: p.adb

```

1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Swap (X, Y : in out Positive) is
5     Tmp : constant Positive := X;
6   begin
7     X := Y;
8     Y := Tmp;
9   end Swap;
10
11  procedure Swap_Indexes (A : in out P_Array; I, J : Natural) is
12  begin
13    Swap (A (I), A (J));
14  end Swap_Indexes;
15
16 end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_05
 MD5: 62a95179572e36443995ff54a2d5ef08

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 p.adb:13:13: medium: formal parameters "X" and "Y" might be aliased (SPARK RM 6.4.
 ↪2)
 gnatprove: unproved check messages considered as errors

GNATprove detects a possible case of aliasing. Unlike the previous example, it has no way of knowing that the two elements A (I) and A (J) are actually distinct when we call Swap. GNATprove issues a check message here instead of an error, giving you the possibility of

justifying the message after review (meaning that you've verified manually that this can't, in fact, occur).

59.8.6 Example #6

We now consider a package declaring a type `Dictionary`, an array containing a word per letter. The procedure `Store` allows us to insert a word at the correct index in a dictionary.

Listing 1781: p.ads

```
1 with Ada.Finalization;
2
3 package P
4   with SPARK_Mode => On
5 is
6   subtype Letter is Character range 'a' .. 'z';
7   type String_Access is new Ada.Finalization.Controlled with record
8     Ptr : access String;
9   end record;
10  type Dictionary is array (Letter) of String_Access;
11
12  procedure Store (D : in out Dictionary; W : String);
13 end P;
```

Listing 1782: p.adb

```
1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Store (D : in out Dictionary; W : String) is
5     First_Letter : constant Letter := W (W'First);
6   begin
7     D (First_Letter).Ptr := new String'(W);
8   end Store;
9 end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_06
MD5: 9175bcd1474e2143462b860c01d8602e

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
p.adb:7:07: error: "String_Access" is not allowed in SPARK (due to controlled_
↳types)
p.adb:7:07: error: violation of aspect SPARK_Mode at line 2
p.adb:7:31: error: borrow or observe of an expression which is not part of stand-
↳alone object or parameter is not allowed in SPARK (SPARK RM 3.10(4))
p.adb:7:31: error: violation of aspect SPARK_Mode at line 2
p.ads:7:09: error: "Controlled" is not allowed in SPARK (due to controlled types)
p.ads:7:09: error: violation of aspect SPARK_Mode at line 4
p.ads:10:04: error: "String_Access" is not allowed in SPARK (due to controlled_
↳types)
p.ads:10:04: error: violation of aspect SPARK_Mode at line 4
gnatprove: error during analysis of data and information flow
```

This code is not correct: controlled types are not part of the SPARK subset. The solution here is to use `SPARK_Mode` to separate the definition of `String_Access` from the rest of the code in a fine grained manner.

59.8.7 Example #7

Here's a new version of the previous example, which we've modified to hide the controlled type inside the private part of package P, using pragma SPARK_Mode (Off) at the start of the private part.

Listing 1783: p.ads

```

1  with Ada.Finalization;
2
3  package P
4    with SPARK_Mode => On
5  is
6    subtype Letter is Character range 'a' .. 'z';
7    type String_Access is private;
8    type Dictionary is array (Letter) of String_Access;
9
10   function New_String_Access (W : String) return String_Access;
11
12   procedure Store (D : in out Dictionary; W : String);
13
14 private
15   pragma SPARK_Mode (Off);
16
17   type String_Access is new Ada.Finalization.Controlled with record
18     Ptr : access String;
19   end record;
20
21   function New_String_Access (W : String) return String_Access is
22     (Ada.Finalization.Controlled with Ptr => new String'(W));
23 end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_07
MD5: cb04206c9734eb95f6444757d005dae2

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

Since the controlled type is defined and used inside of a part of the code ignored by GNAT-prove, this code is correct.

59.8.8 Example #8

Let's put together the new spec for package P with the body of P seen previously.

Listing 1784: p.ads

```

1  with Ada.Finalization;
2
3  package P
4    with SPARK_Mode => On
5  is
6    subtype Letter is Character range 'a' .. 'z';
7    type String_Access is private;
8    type Dictionary is array (Letter) of String_Access;
9
10   function New_String_Access (W : String) return String_Access;
```

(continues on next page)

(continued from previous page)

```

11
12     procedure Store (D : in out Dictionary; W : String);
13
14 private
15     pragma SPARK_Mode (Off);
16
17     type String_Access is new Ada.Finalization.Controlled with record
18         Ptr : access String;
19     end record;
20
21     function New_String_Access (W : String) return String_Access is
22         (Ada.Finalization.Controlled with Ptr => new String'(W));
23 end P;

```

Listing 1785: p.adb

```

1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Store (D : in out Dictionary; W : String) is
5       First_Letter : constant Letter := W (W'First);
6   begin
7       D (First_Letter) := New_String_Access (W);
8   end Store;
9 end P;

```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_08
MD5: dacb2d50d0ddc6c620ee9945cb819369

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
p.adb:1:01: error: incorrect application of SPARK_Mode at /vagrant/frontend/dist/
↳ test_output/projects/Courses/Intro_To_Spark/Overview/Example_08/
↳ dacb2d50d0ddc6c620ee9945cb819369/main_spark.adc:12
p.adb:1:01: error: value Off was set for SPARK_Mode on "P" at p.ads:15
p.adb:2:08: error: incorrect use of SPARK_Mode
p.adb:2:08: error: value Off was set for SPARK_Mode on "P" at p.ads:15
gnatprove: error during generation of Global contracts

```

The body of Store doesn't actually use any construct that's not in the SPARK subset, but we nevertheless can't set SPARK_Mode to On for P's body because it has visibility to P's private part, which is not in SPARK, even if we don't use it.

59.8.9 Example #9

Next, we moved the declaration and the body of the procedure Store to another package named Q.

Listing 1786: p.ads

```

1 with Ada.Finalization;
2
3 package P
4   with SPARK_Mode => On
5 is
6   subtype Letter is Character range 'a' .. 'z';

```

(continues on next page)

(continued from previous page)

```

7  type String_Access is private;
8  type Dictionary is array (Letter) of String_Access;
9
10 function New_String_Access (W : String) return String_Access;
11
12 private
13   pragma SPARK_Mode (Off);
14
15   type String_Access is new Ada.Finalization.Controlled with record
16     Ptr : access String;
17   end record;
18
19   function New_String_Access (W : String) return String_Access is
20     (Ada.Finalization.Controlled with Ptr => new String'(W));
21 end P;
```

Listing 1787: q.ads

```

1  with P; use P;
2  package Q
3    with SPARK_Mode => On
4  is
5    procedure Store (D : in out Dictionary; W : String);
6  end Q;
```

Listing 1788: q.adb

```

1  package body Q
2    with SPARK_Mode => On
3  is
4    procedure Store (D : in out Dictionary; W : String) is
5      First_Letter : constant Letter := W (W'First);
6    begin
7      D (First_Letter) := New_String_Access (W);
8    end Store;
9  end Q;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_09
MD5: b397e82987c100de5a53ede16fbef37f

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

And now everything is fine: we've managed to retain the use of the controlled type while having most of our code in the SPARK subset so GNATprove is able to analyze it.

59.8.10 Example #10

Our final example is a package with two functions to search for the value 0 inside an array A. The first raises an exception if 0 isn't found in A while the other simply returns 0 in that case.

Listing 1789: p.ads

```
1 package P
2   with SPARK_Mode => On
3 is
4   type N_Array is array (Positive range <>) of Natural;
5   Not_Found : exception;
6
7   function Search_Zero_P (A : N_Array) return Positive;
8
9   function Search_Zero_N (A : N_Array) return Natural;
10 end P;
```

Listing 1790: p.adb

```
1 package body P
2   with SPARK_Mode => On
3 is
4   function Search_Zero_P (A : N_Array) return Positive is
5   begin
6     for I in A'Range loop
7       if A (I) = 0 then
8         return I;
9       end if;
10    end loop;
11    raise Not_Found;
12 end Search_Zero_P;
13
14 function Search_Zero_N (A : N_Array) return Natural
15 with SPARK_Mode => Off is
16 begin
17   return Search_Zero_P (A);
18 exception
19   when Not_Found => return 0;
20 end Search_Zero_N;
21 end P;
```

Code block metadata

Project: Courses.Intro_To_Spark.Overview.Example_10
MD5: 4b9656698abd42cebc72817f8a00637

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
p.adb:11:07: medium: unexpected exception might be raised
gnatprove: unproved check messages considered as errors

This code is perfectly correct, despite the use of exception handling, because we've carefully isolated this non-SPARK feature in a function body marked with a `SPARK_Mode` of `Off` so it's ignored by GNATprove. However, GNATprove tries to show that `Not_Found` is never raised in `Search_Zero_P`, producing a message about a possible exception being raised. Looking at `Search_Zero_N`, it's indeed likely that an exception is meant to be raised in some cases, which means you need to verify that `Not_Found` is only raised when appropriate using other methods such as peer review or testing.

FLOW ANALYSIS

In this section we present the flow analysis capability provided by the GNATprove tool, a critical tool for using SPARK.

60.1 What does flow analysis do?

Flow analysis concentrates primarily on variables. It models how information flows through them during a subprogram's execution, connecting the final values of variables to their initial values. It analyzes global variables declared at library level, local variables, and formal parameters of subprograms.

Nesting of subprograms creates what we call *scope variables*: variables declared locally to an enclosing unit. From the perspective of a nested subprogram, scope variables look very much like global variables

Flow analysis is usually fast, roughly as fast as compilation. It detects various types of errors and finds violations of some SPARK legality rules, such as the absence of aliasing and freedom of expressions from side-effects. We discussed these rules in the [SPARK Overview](#) (page 1413).

Flow analysis is *sound*: if it doesn't detect any errors of a type it's supposed to detect, we know for sure there are no such errors.

60.2 Errors Detected

60.2.1 Uninitialized Variables

We now present each class of errors detected by flow analysis. The first is the reading of an uninitialized variable. This is nearly always an error: it introduces non-determinism and breaks the type system because the value of an uninitialized variable may be outside the range of its subtype. For these reasons, SPARK requires every variable to be initialized before being read.

Flow analysis is responsible for ensuring that SPARK code always fulfills this requirement. For example, in the function `Max_Array` shown below, we've neglected to initialize the value of `Max` prior to entering the loop. As a consequence, the value read by the condition of the `if` statement may be uninitialized. Flow analysis detects and reports this error.

Listing 1791: `show_uninitialized.ads`

```
1 package Show_Uninitialized is
2
3   type Array_Of_Naturals is array (Integer range <>) of Natural;
4
5   function Max_Array (A : Array_Of_Naturals) return Natural;
```

(continues on next page)

(continued from previous page)

```

6
7 end Show_Uninitialized;

```

Listing 1792: show_uninitialized.adb

```

1 package body Show_Uninitialized is
2
3   function Max_Array (A : Array_Of_Naturals) return Natural is
4     Max : Natural;
5   begin
6     for I in A'Range loop
7       if A (I) > Max then -- Here Max may not be initialized
8         Max := A (I);
9       end if;
10    end loop;
11    return Max;
12  end Max_Array;
13
14 end Show_Uninitialized;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Uninitialized
MD5: 82fe32cbe33e25bac5466f86ee2e03c4

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_uninitialized.adb:7:21: medium: "Max" might not be initialized
show_uninitialized.adb:11:14: medium: "Max" might not be initialized
gnatprove: unproved check messages considered as errors

Note

For more details on how flow analysis verifies data initialization, see the [SPARK User's Guide](#)³⁹⁴.

60.2.2 Ineffective Statements

Ineffective statements are different than dead code: they're executed, and often even modify the value of variables, but have no effect on any of the subprogram's visible outputs: parameters, global variables or the function result. Ineffective statements should be avoided because they make the code less readable and more difficult to maintain.

More importantly, they're often caused by errors in the program: the statement may have been written for some purpose, but isn't accomplishing that purpose. These kinds of errors can be difficult to detect in other ways.

For example, the subprograms Swap1 and Swap2 shown below don't properly swap their two parameters X and Y. This error caused a statement to be ineffective. That ineffective statement is not an error in itself, but flow analysis produces a warning since it can be indicative of an error, as it is here.

³⁹⁴ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#data-initialization-policy

Listing 1793: show_ineffective_statements.ads

```

1 package Show_Ineffective_Statements is
2
3     type T is new Integer;
4
5     procedure Swap1 (X, Y : in out T);
6     procedure Swap2 (X, Y : in out T);
7
8 end Show_Ineffective_Statements;
```

Listing 1794: show_ineffective_statements.adb

```

1 package body Show_Ineffective_Statements is
2
3     procedure Swap1 (X, Y : in out T) is
4         Tmp : T;
5     begin
6         Tmp := X; -- This statement is ineffective
7         X := Y;
8         Y := X;
9     end Swap1;
10
11     Tmp : T := 0;
12
13     procedure Swap2 (X, Y : in out T) is
14         Temp : T := X; -- This variable is unused
15     begin
16         X := Y;
17         Y := Temp;
18     end Swap2;
19
20 end Show_Ineffective_Statements;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Ineffective_Statements
MD5: 473a9215e9e98bd25147998d43847a12

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_ineffective_statements.ads:5:21: warning: unused initial value of "X"
show_ineffective_statements.ads:6:21: warning: unused initial value of "X"
```

So far, we've seen examples where flow analysis warns about ineffective statements and unused variables.

60.2.3 Incorrect Parameter Mode

Parameter modes are an important part of documenting the usage of a subprogram and affect the code generated for that subprogram. Flow analysis checks that each specified parameter mode corresponds to the usage of that parameter in the subprogram's body. It checks that an **in** parameter is never modified, either directly or through a subprogram call, checks that the initial value of an **out** parameter is never read in the subprogram (since it may not be defined on subprogram entry), and warns when an **in out** parameter isn't modified or when its initial value isn't used. All of these may be signs of an error.

We see an example below. The subprogram `Swap` is incorrect and GNATprove warns about an input which isn't read:

Listing 1795: `show_incorrect_param_mode.ads`

```
1 package Show_Incorrect_Param_Mode is
2
3   type T is new Integer;
4
5   procedure Swap (X, Y : in out T);
6
7 end Show_Incorrect_Param_Mode;
```

Listing 1796: `show_incorrect_param_mode.adb`

```
1 package body Show_Incorrect_Param_Mode is
2
3   procedure Swap (X, Y : in out T) is
4     Tmp : T := X;
5   begin
6     Y := X; -- The initial value of Y is not used
7     X := Tmp; -- Y is computed to be an out parameter
8   end Swap;
9
10 end Show_Incorrect_Param_Mode;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Incorrect_Param_Mode
MD5: 1e33dbf461daab0daed01c83025232fc

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_incorrect_param_mode.ads:5:23: warning: unused initial value of "Y"

In SPARK, unlike Ada, you should declare an **out** parameter to be **in out** if it's not modified on every path, in which case its value may depend on its initial value. SPARK is stricter than Ada to allow more static detection of errors. This table summarizes SPARK's valid parameter modes as a function of whether reads and writes are done to the parameter.

Initial value read	Written on some path	Written on every path	Parameter mode
X			in
X	X		in out
X		X	in out
	X		in out
		X	out

60.3 Additional Verifications

60.3.1 Global Contracts

So far, none of the verifications we've seen require you to write any additional annotations. However, flow analysis also checks flow annotations that you write. In SPARK, you can specify the set of global and scoped variables accessed or modified by a subprogram. You do this using a contract named `Global`.

When you specify a Global contract for a subprogram, flow analysis checks that it's both correct and complete, meaning that no variables other than those stated in the contract are accessed or modified, either directly or through a subprogram call, and that all those listed are accessed or modified. For example, we may want to specify that the function `Get_Value_Of_X` reads the value of the global variable `X` and doesn't access any other global variable. If we do this through a comment, as is usually done in other languages, GNATprove can't verify that the code complies with this specification:

```
package Show_Global_Contracts is

  X : Natural := 0;

  function Get_Value_Of_X return Natural;
  -- Get_Value_Of_X reads the value of the global variable X

end Show_Global_Contracts;
```

You write global contracts as part of the subprogram specification. In addition to their value in flow analysis, they also provide useful information to users of a subprogram. The value you specify for the Global aspect is an aggregate-like list of global variable names, grouped together according to their mode.

In the example below, the procedure `Set_X_To_Y_Plus_Z` reads both `Y` and `Z`. We indicate this by specifying them as the value for Input. It also writes `X`, which we specify using Output. Since `Set_X_To_X_Plus_Y` both writes `X` and reads its initial value, `X`'s mode is `In_Out`. Like parameters, if no mode is specified in a Global aspect, the default is Input. We see this in the case of the declaration of `Get_Value_Of_X`. Finally, if a subprogram, such as `Incr_Parameter_X`, doesn't reference any global variables, you set the value of the global contract to `null`.

Listing 1797: show_global_contracts.ads

```
1 package Show_Global_Contracts is
2
3   X, Y, Z : Natural := 0;
4
5   procedure Set_X_To_Y_Plus_Z with
6     Global => (Input => (Y, Z), -- reads values of Y and Z
7               Output => X);    -- modifies value of X
8
9   procedure Set_X_To_X_Plus_Y with
10    Global => (Input => Y, -- reads value of Y
11              In_Out => X); -- modifies value of X and
12                          -- also reads its initial value
13
14   function Get_Value_Of_X return Natural with
15     Global => X; -- reads the value of the global variable X
16
17   procedure Incr_Parameter_X (X : in out Natural) with
18     Global => null; -- do not reference any global variable
19
20 end Show_Global_Contracts;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Global_Contracts
MD5: 2cbf90f2d27b6b0043a2e29449e79df9

Prover output

Phase 1 of 3: generation of data representation information ...

(continues on next page)

(continued from previous page)

Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...

Note

For more details on global contracts, see the [SPARK User's Guide](#)³⁹⁵.

60.3.2 Depends Contracts

You may also supply a Depends contract for a subprogram to specify dependencies between its inputs and outputs. These dependencies include not only global variables but also parameters and the function's result. When you supply a Depends contract for a subprogram, flow analysis checks that it's correct and complete, that is, for each dependency you list, the variable depends on those listed and on no others.

For example, you may want to say that the new value of each parameter of `Swap`, shown below, depends only on the initial value of the other parameter and that the value of `X` after the return of `Set_X_To_Zero` doesn't depend on any global variables. If you indicate this through a comment, as you often do in other languages, GNATprove can't verify that this is actually the case.

```
package Show_Depends_Contracts is

  type T is new Integer;

  procedure Swap (X, Y : in out T);
  -- The value of X (resp. Y) after the call depends only
  -- on the value of Y (resp. X) before the call

  X : Natural;
  procedure Set_X_To_Zero;
  -- The value of X after the call depends on no input

end Show_Depends_Contracts;
```

Like Global contracts, you specify a Depends contract in subprogram declarations using an aspect. Its value is a list of one or more dependency relations between the outputs and inputs of the subprogram. Each relation is represented as two lists of variable names separated by an arrow. On the left of each arrow are variables whose final value depends on the initial value of the variables you list on the right.

For example, here we indicate that the final value of each parameter of `Swap` depends only on the initial value of the other parameter. If the subprogram is a function, we list its result as an output, using the `Result` attribute, as we do for `Get_Value_Of_X` below.

Listing 1798: `show_depends_contracts.ads`

```
1 package Show_Depends_Contracts is
2
3   type T is new Integer;
4
5   X, Y, Z : T := 0;
6
7   procedure Swap (X, Y : in out T) with
8     Depends => (X => Y,
```

(continues on next page)

³⁹⁵ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#data-dependencies

(continued from previous page)

```

9      -- X depends on the initial value of Y
10     Y => X);
11     -- Y depends on the initial value of X
12
13     function Get_Value_Of_X return T with
14         Depends => (Get_Value_Of_X'Result => X);
15         -- result depends on the initial value of X
16
17     procedure Set_X_To_Y_Plus_Z with
18         Depends => (X => (Y, Z));
19         -- X depends on the initial values of Y and Z
20
21     procedure Set_X_To_X_Plus_Y with
22         Depends => (X =>+ Y);
23         -- X depends on Y and X's initial value
24
25     procedure Do_Nothing (X : T) with
26         Depends => (null => X);
27         -- no output is affected by X
28
29     procedure Set_X_To_Zero with
30         Depends => (X => null);
31         -- X depends on no input
32
33 end Show_Depends_Contracts;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Depends_Contracts
MD5: 290866c4208b6deff717a402bc2aef34

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

Often, the final value of a variable depends on its own initial value. You can specify this in a concise way using the + character, as we did in the specification of `Set_X_To_X_Plus_Y` above. If there's more than one variable on the left of the arrow, a + means each variables depends on itself, not that they all depend on each other. You can write the corresponding dependency with `(=> +)` or without `(=>+)` whitespace.

If you have a program where an input isn't used to compute the final value of any output, you express that by writing `null` on the left of the dependency relation, as we did for the `Do_Nothing` subprogram above. You can only write one such dependency relation, which lists all unused inputs of the subprogram, and it must be written last. Such an annotation also silences flow analysis' warning about unused parameters. You can also write `null` on the right of a dependency relation to indicate that an output doesn't depend on any input. We do that above for the procedure `Set_X_To_Zero`.

Note

For more details on depends contracts, see the [SPARK User's Guide](#)³⁹⁶.

³⁹⁶ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#flow-dependencies

60.4 Shortcomings

60.4.1 Modularity

Flow analysis is sound, meaning that if it doesn't output a message on some analyzed SPARK code, you can be assured that none of the errors it tests for can occur in that code. On the other hand, flow analysis often issues messages when there are, in fact, no errors. The first, and probably most common reason for this relates to modularity.

To scale flow analysis to large projects, verifications are usually done on a per-subprogram basis, including detection of uninitialized variables. To analyze this modularly, flow analysis needs to assume the initialization of inputs on subprogram entry and modification of outputs during subprogram execution. Therefore, each time a subprogram is called, flow analysis checks that global and parameter inputs are initialized and each time a subprogram returns, it checks that global and parameter outputs were modified.

This can produce error messages on perfectly correct subprograms. An example is `Set_X_To_Y_Plus_Z` below, which only sets its `out` parameter `X` when `Overflow` is `False`.

Listing 1799: `set_x_to_y_plus_z.adb`

```

1  procedure Set_X_To_Y_Plus_Z
2    (Y, Z      : Natural;
3     X        : out Natural;
4     Overflow  : out Boolean)
5  is
6  begin
7    if Natural'Last - Z < Y then
8      Overflow := True; -- X should be initialized on every path
9    else
10     Overflow := False;
11     X := Y + Z;
12   end if;
13 end Set_X_To_Y_Plus_Z;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Set_X_To_Y_Plus_Z
MD5: be47cd769d2a7267c0bd1bb2ef0d6328

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
set_x_to_y_plus_z.adb:3:04: medium: "X" might not be initialized in "Set_X_To_Y_
↳ Plus_Z" [reason for check: OUT parameter should be initialized on return]
↳ [possible fix: initialize "X" on all paths or make "X" an IN OUT parameter]
gnatprove: unproved check messages considered as errors
```

The message means that flow analysis wasn't able to verify that the program didn't read an uninitialized variable. To solve this problem, you can either set `X` to a dummy value when there's an overflow or manually verify that `X` is never used after a call to `Set_X_To_Y_Plus_Z` that returned `True` as the value of `Overflow`.

60.4.2 Composite Types

Another common cause of false alarms is caused by the way flow analysis handles composite types. Let's start with arrays.

Flow analysis treats an entire array as single object instead of one object per element, so it considers modifying a single element to be a modification of the array as a whole.

Obviously, this makes reasoning about which global variables are accessed less precise and hence the dependencies of those variables are also less precise. This also affects the ability to accurately detect reads of uninitialized data.

It's sometimes impossible for flow analysis to determine if an entire array object has been initialized. For example, after we write code to initialize every element of an unconstrained array A in chunks, we may still receive a message from flow analysis claiming that the array isn't initialized. To resolve this issue, you can either use a simpler loop over the full range of the array, or (even better) an aggregate assignment, or, if that's not possible, verify initialization of the object manually.

Listing 1800: show_composite_types_shortcoming.ads

```

1 package Show_Composite_Types_Shortcoming is
2
3   type T is array (Natural range <>) of Integer;
4
5   procedure Init_Chunks (A : out T);
6   procedure Init_Loop (A : out T);
7   procedure Init_Aggregate (A : out T);
8
9 end Show_Composite_Types_Shortcoming;
```

Listing 1801: show_composite_types_shortcoming.adb

```

1 package body Show_Composite_Types_Shortcoming is
2
3   procedure Init_Chunks (A : out T) is
4   begin
5     A (A'First) := 0;
6     for I in A'First + 1 .. A'Last loop
7       A (I) := 0;
8     end loop;
9     -- flow analysis doesn't know that A is initialized
10  end Init_Chunks;
11
12  procedure Init_Loop (A : out T) is
13  begin
14    for I in A'Range loop
15      A (I) := 0;
16    end loop;
17    -- flow analysis knows that A is initialized
18  end Init_Loop;
19
20  procedure Init_Aggregate (A : out T) is
21  begin
22    A := (others => 0);
23    -- flow analysis knows that A is initialized
24  end Init_Aggregate;
25
26 end Show_Composite_Types_Shortcoming;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Composite_Types_Shortcoming
MD5: a366dcdd141191466027b2b928560c5e

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

(continues on next page)

(continued from previous page)

```
show_composite_types_shortcoming.ads:5:27: medium: "A" might not be initialized in
↳ "Init_Chunks" [reason for check: OUT parameter should be fully initialized on
↳ return] [possible fix: initialize "A" on all paths, make "A" an IN OUT parameter,
↳ or annotate it with aspect Relaxed_Initialization]
gnatprove: unproved check messages considered as errors
```

Flow analysis is more precise on record objects because it tracks the value of each component of a record separately within a single subprogram. So when a record object is initialized by successive assignments of its components, flow analysis knows that the entire object is initialized. However, record objects are still treated as single objects when analyzed as an input or output of a subprogram.

Listing 1802: show_record_flow_analysis.ads

```
1 package Show_Record_Flow_Analysis is
2
3     type Rec is record
4         F1 : Natural;
5         F2 : Natural;
6     end record;
7
8     procedure Init (R : out Rec);
9
10 end Show_Record_Flow_Analysis;
```

Listing 1803: show_record_flow_analysis.adb

```
1 package body Show_Record_Flow_Analysis is
2
3     procedure Init (R : out Rec) is
4     begin
5         R.F1 := 0;
6         R.F2 := 0;
7         -- R is initialized
8     end Init;
9
10 end Show_Record_Flow_Analysis;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Record_Flow_Analysis_1
MD5: 24cd553b87b737536912b1bb780f6402

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_record_flow_analysis.ads:8:20: info: initialization of "R" proved
```

Flow analysis complains when a procedure call initializes only some components of a record object. It'll notify you of uninitialized components, as we see in subprogram Init_F2 below.

Listing 1804: show_record_flow_analysis.ads

```
1 package Show_Record_Flow_Analysis is
2
3     type Rec is record
4         F1 : Natural;
5         F2 : Natural;
```

(continues on next page)

(continued from previous page)

```

6   end record;
7
8   procedure Init (R : out Rec);
9   procedure Init_F2 (R : in out Rec);
10
11 end Show_Record_Flow_Analysis;

```

Listing 1805: show_record_flow_analysis.adb

```

1  package body Show_Record_Flow_Analysis is
2
3      procedure Init_F2
4      (R : in out Rec) is
5      begin
6          R.F2 := 0;
7      end Init_F2;
8
9      procedure Init (R : out Rec) is
10     begin
11         R.F1 := 0;
12         Init_F2 (R); -- R should be initialized before this call
13     end Init;
14
15 end Show_Record_Flow_Analysis;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Record_Flow_Analysis_2
MD5: efeecb787bf9d68977ed9701689cd6c4

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_record_flow_analysis.adb:12:16: high: "R.F2" is not initialized
gnatprove: unproved check messages considered as errors

60.4.3 Value Dependency

Flow analysis is not value-dependent: it never reasons about the values of expressions, only whether they have been set to some value or not. As a consequence, if some execution path in a subprogram is impossible, but the impossibility can only be determined by looking at the values of expressions, flow analysis still considers that path feasible and may emit messages based on it believing that execution along such a path is possible.

For example, in the version of `Absolute_Value` below, flow analysis computes that `R` is uninitialized on a path that enters neither of the two conditional statements. Because it doesn't consider values of expressions, it can't know that such a path is impossible.

Listing 1806: absolute_value.adb

```

1  procedure Absolute_Value
2  (X : Integer;
3   R : out Natural)
4  is
5  begin
6      if X < 0 then
7          R := -X;

```

(continues on next page)

(continued from previous page)

```

8   end if;
9   if X >= 0 then
10      R := X;
11   end if;
12   -- flow analysis doesn't know that R is initialized
13 end Absolute_Value;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Absolute_Value_1
MD5: 69c233d22afdfdac679bf379b353a8d4

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
absolute_value.adb:3:04: medium: "R" might not be initialized in "Absolute_Value"
↳[reason for check: OUT parameter should be initialized on return] [possible fix:
↳initialize "R" on all paths or make "R" an IN OUT parameter]
gnatprove: unproved check messages considered as errors

To avoid this problem, you should make the control flow explicit, as in this second version of Absolute_Value:

Listing 1807: absolute_value.adb

```

1  procedure Absolute_Value
2  (X :      Integer;
3   R : out Natural)
4  is
5  begin
6      if X < 0 then
7          R := -X;
8      else
9          R := X;
10     end if;
11     -- flow analysis knows that R is initialized
12 end Absolute_Value;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Absolute_Value_2
MD5: 9c773547f81e82a7aa1b45132b105937

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

60.4.4 Contract Computation

The final cause of unexpected flow messages that we'll discuss also comes from inaccuracy in computations of contracts. As we explained earlier, both Global and Depends contracts are optional, but GNATprove uses their data for some of its analysis.

For example, flow analysis can't detect reads from uninitialized variables without knowing the set of variables accessed. It needs to analyze and check both the Depends contracts you wrote for a subprogram and those you wrote for callers of that subprogram. Since

each flow contract on a subprogram depends on the flow contracts of all the subprograms called inside its body, this computation can often be quite time-consuming. Therefore, flow analysis sometimes trades-off the precision of this computation against the time a more precise computation would take.

This is the case for Depends contracts, where flow analysis simply assumes the worst, that each subprogram's output depends on all of that subprogram's inputs. To avoid this assumption, all you have to do is supply contracts when default ones are not precise enough. You may also want to supply Global contracts to further speed up flow analysis on larger programs.

60.5 Code Examples / Pitfalls

60.5.1 Example #1

The procedure `Search_Array` searches for an occurrence of element `E` in an array `A`. If it finds one, it stores the index of the element in `Result`. Otherwise, it sets `Found` to **False**.

Listing 1808: `show_search_array.ads`

```

1 package Show_Search_Array is
2
3   type Array_Of_Positives is array (Natural range <>) of Positive;
4
5   procedure Search_Array
6     (A      : Array_Of_Positives;
7      E      : Positive;
8      Result : out Integer;
9      Found  : out Boolean);
10
11 end Show_Search_Array;
```

Listing 1809: `show_search_array.adb`

```

1 package body Show_Search_Array is
2
3   procedure Search_Array
4     (A      : Array_Of_Positives;
5      E      : Positive;
6      Result : out Integer;
7      Found  : out Boolean) is
8   begin
9     for I in A'Range loop
10      if A (I) = E then
11        Result := I;
12        Found  := True;
13        return;
14      end if;
15    end loop;
16    Found := False;
17  end Search_Array;
18
19 end Show_Search_Array;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_01
MD5: d2a27a5bde247767e2f6cd2d42a2d629

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_search_array.ads:8:07: medium: "Result" might not be initialized in "Search_
↳Array" [reason for check: OUT parameter should be initialized on return]_
↳[possible fix: initialize "Result" on all paths or make "Result" an IN OUT_
↳parameter]
gnatprove: unproved check messages considered as errors
```

GNATprove produces a message saying that Result is possibly uninitialized on return. There are perfectly legal uses of the function Search_Array, but flow analysis detects that Result is not initialized on the path that falls through from the loop. Even though this program is correct, you shouldn't ignore the message: it means flow analysis cannot guarantee that Result is always initialized at the call site and so assumes any read of Result at the call site will read initialized data. Therefore, you should either initialize Result when Found is false, which silences flow analysis, or verify this assumption at each call site by other means.

60.5.2 Example #2

To avoid the message previously issued by GNATprove, we modify Search_Array to raise an exception when E isn't found in A:

Listing 1810: show_search_array.ads

```
1 package Show_Search_Array is
2
3     type Array_Of_Positives is array (Natural range <>) of Positive;
4
5     Not_Found : exception;
6
7     procedure Search_Array
8         (A      : Array_Of_Positives;
9          E      : Positive;
10           Result : out Integer);
11 end Show_Search_Array;
```

Listing 1811: show_search_array.adb

```
1 package body Show_Search_Array is
2
3     procedure Search_Array
4         (A      : Array_Of_Positives;
5          E      : Positive;
6          Result : out Integer) is
7     begin
8         for I in A'Range loop
9             if A (I) = E then
10                 Result := I;
11                 return;
12             end if;
13         end loop;
14         raise Not_Found;
15     end Search_Array;
16
17 end Show_Search_Array;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_02
 MD5: fa159faeb68974b1af3de2112e086b16

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: flow analysis and proof ...
 show_search_array.adb:14:07: medium: unexpected exception might be raised
 gnatprove: unproved check messages considered as errors

Flow analysis doesn't emit any messages in this case, meaning it can verify that Result can't be read in SPARK code while uninitialized. But why is that, since Result is still not initialized when E is not in A? This is because the exception, Not_Found, can never be caught within SPARK code (SPAK doesn't allow exception handlers). However, the GNATprove tool also tries to ensure the absence of runtime errors in SPARK code, so tries to prove that Not_Found is never raised. When it can't do that here, it produces a different message.

60.5.3 Example #3

In this example, we're using a discriminated record for the result of Search_Array instead of conditionally raising an exception. By using such a structure, the place to store the index at which E was found exists only when E was indeed found. So if it wasn't found, there's nothing to be initialized.

Listing 1812: show_search_array.ads

```

1 package Show_Search_Array is
2
3   type Array_Of_Positives is array (Natural range <>) of Positive;
4
5   type Search_Result (Found : Boolean := False) is record
6     case Found is
7       when True =>
8         Content : Integer;
9       when False => null;
10    end case;
11  end record;
12
13  procedure Search_Array
14    (A      : Array_Of_Positives;
15     E      : Positive;
16     Result : out Search_Result)
17  with Pre => not Result'Constrained;
18
19 end Show_Search_Array;
```

Listing 1813: show_search_array.adb

```

1 package body Show_Search_Array is
2
3   procedure Search_Array
4     (A      : Array_Of_Positives;
5      E      : Positive;
6      Result : out Search_Result) is
7   begin
8     for I in A'Range loop
9       if A (I) = E then
10        Result := (Found  => True,
11                  Content => I);
```

(continues on next page)

(continued from previous page)

```

12         return;
13     end if;
14 end loop;
15 Result := (Found => False);
16 end Search_Array;
17
18 end Show_Search_Array;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_03
MD5: 1d5ec5d78185fd75499b90b3d21f8ae2

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_search_array.adb:10:20: info: discriminant check proved
show_search_array.adb:15:14: info: discriminant check proved
show_search_array.ads:16:07: info: initialization of "Result" proved
```

This example is correct and flow analysis doesn't issue any message: it can verify both that no uninitialized variables are read in `Search_Array`'s body, and that all its outputs are set on return. We've used the attribute `Constrained` in the precondition of `Search_Array` to indicate that the value of the `Result` in argument can be set to any variant of the record type `Search_Result`, specifically to either the variant where `E` was found and where it wasn't.

60.5.4 Example #4

The function `Size_Of_Biggest_Increasing_Sequence` is supposed to find all sequences within its parameter `A` that contain elements with increasing values and returns the length of the longest one. To do this, it calls a nested procedure `Test_Index` iteratively on all the elements of `A`. `Test_Index` checks if the sequence is still increasing. If so, it updates the largest value seen so far in this sequence. If not, it means it's found the end of a sequence, so it computes the size of that sequence and stores it in `Size_Of_Seq`.

Listing 1814: `show_biggest_increasing_sequence.ads`

```

1 package Show_Biggest_Increasing_Sequence is
2
3     type Array_Of_Positives is array (Integer range <>) of Positive;
4
5     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
6         return Natural;
7
8 end Show_Biggest_Increasing_Sequence;
```

Listing 1815: `show_biggest_increasing_sequence.adb`

```

1 package body Show_Biggest_Increasing_Sequence is
2
3     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
4         return Natural
5     is
6         Max          : Natural;
7         End_Of_Seq   : Boolean;
8         Size_Of_Seq  : Natural;
```

(continues on next page)

(continued from previous page)

```

9      Beginning    : Integer;
10
11     procedure Test_Index (Current_Index : Integer) is
12     begin
13         if A (Current_Index) >= Max then
14             Max := A (Current_Index);
15             End_Of_Seq := False;
16         else
17             Max           := 0;
18             End_Of_Seq    := True;
19             Size_Of_Seq   := Current_Index - Beginning;
20             Beginning     := Current_Index;
21         end if;
22     end Test_Index;
23
24     Biggest_Seq : Natural := 0;
25
26     begin
27         for I in A'Range loop
28             Test_Index (I);
29             if End_Of_Seq then
30                 Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
31             end if;
32         end loop;
33         return Biggest_Seq;
34     end Size_Of_Biggest_Increasing_Sequence;
35
36 end Show_Biggest_Increasing_Sequence;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_04
MD5: e6083665827d9dee4e00bdce4c1e962f

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:13:34: medium: "Max" might not be initialized,
↳ in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:19:44: medium: "Beginning" might not be
↳ initialized, in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:30:41: medium: "Size_Of_Seq" might not be
↳ initialized
gnatprove: unproved check messages considered as errors

```

However, this example is not correct. Flow analysis emits messages for `Test_Index` stating that `Max`, `Beginning`, and `Size_Of_Seq` should be initialized before being read. Indeed, when you look carefully, you see that both `Max` and `Beginning` are missing initializations because they are read in `Test_Index` before being written. As for `Size_Of_Seq`, we only read its value when `End_Of_Seq` is true, so it actually can't be read before being written, but flow analysis isn't able to verify its initialization by using just flow information.

The call to `Test_Index` is automatically inlined by GNATprove, which leads to another messages above. If GNATprove couldn't inline the call to `Test_Index`, for example if it was defined in another unit, the same messages would be issued on the call to `Test_Index`.

60.5.5 Example #5

In the following example, we model permutations as arrays where the element at index *I* is the position of the *I*'th element in the permutation. The procedure `Init` initializes a permutation to the identity, where the *I*'th elements is at the *I*'th position. `Cyclic_Permutation` calls `Init` and then swaps elements to construct a cyclic permutation.

Listing 1816: show_permutation.ads

```

1 package Show_Permutation is
2
3   type Permutation is array (Positive range <>) of Positive;
4
5   procedure Swap (A : in out Permutation;
6                  I, J : Positive);
7
8   procedure Init (A : out Permutation);
9
10  function Cyclic_Permutation (N : Natural) return Permutation;
11
12 end Show_Permutation;
```

Listing 1817: show_permutation.adb

```

1 package body Show_Permutation is
2
3   procedure Swap (A : in out Permutation;
4                  I, J : Positive)
5   is
6     Tmp : Positive := A (I);
7   begin
8     A (I) := A (J);
9     A (J) := Tmp;
10  end Swap;
11
12  procedure Init (A : out Permutation) is
13  begin
14    A (A'First) := A'First;
15    for I in A'First + 1 .. A'Last loop
16      A (I) := I;
17    end loop;
18  end Init;
19
20  function Cyclic_Permutation (N : Natural) return Permutation is
21    A : Permutation (1 .. N);
22  begin
23    Init (A);
24    for I in A'First .. A'Last - 1 loop
25      Swap (A, I, I + 1);
26    end loop;
27    return A;
28  end Cyclic_Permutation;
29
30 end Show_Permutation;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_05
MD5: 219b06617c636c18543128d77f90fcee

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 show_permutation.ads:8:20: medium: "A" might not be initialized in "Init" [reason:
 ↳for check: OUT parameter should be fully initialized on return] [possible fix:
 ↳initialize "A" on all paths, make "A" an IN OUT parameter or annotate it with
 ↳aspect Relaxed_Initialization]
 gnatprove: unproved check messages considered as errors

This program is correct. However, flow analysis will nevertheless still emit messages because it can't verify that every element of A is initialized by the loop in Init. This message is a false alarm. You can either ignore it or justify it safely.

60.5.6 Example #6

This program is the same as the previous one except that we've changed the mode of A in the specification of Init to **in out** to avoid the message from flow analysis on array assignment.

Listing 1818: show_permutation.ads

```

1 package Show_Permutation is
2
3     type Permutation is array (Positive range <>) of Positive;
4
5     procedure Swap (A      : in out Permutation;
6                     I, J : Positive);
7
8     procedure Init (A : in out Permutation);
9
10    function Cyclic_Permutation (N : Natural) return Permutation;
11
12 end Show_Permutation;
```

Listing 1819: show_permutation.adb

```

1 package body Show_Permutation is
2
3     procedure Swap (A      : in out Permutation;
4                     I, J : Positive)
5     is
6         Tmp : Positive := A (I);
7     begin
8         A (I) := A (J);
9         A (J) := Tmp;
10    end Swap;
11
12    procedure Init (A : in out Permutation) is
13    begin
14        A (A'First) := A'First;
15        for I in A'First + 1 .. A'Last loop
16            A (I) := I;
17        end loop;
18    end Init;
19
20    function Cyclic_Permutation (N : Natural) return Permutation is
21        A : Permutation (1 .. N);
22    begin
23        Init (A);
24        for I in A'First .. A'Last - 1 loop
```

(continues on next page)

(continued from previous page)

```

25     Swap (A, I, I + 1);
26   end loop;
27   return A;
28 end Cyclic_Permutation;
29
30 end Show_Permutation;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_06
MD5: 61406d9a66dda71630c74c12f3d67936

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_permutation.adb:23:13: high: "A" is not initialized
gnatprove: unproved check messages considered as errors
```

This program is not correct. Changing the mode of a parameter that should really be **out** to **in out** to silence a false alarm is not a good idea. Not only does this obfuscate the specification of Init, but flow analysis emits a message on the procedure where A is not initialized, as shown by the message in Cyclic_Permutation.

60.5.7 Example #7

Incr_Step_Function takes an array A as an argument and iterates through A to increment every element by the value of Increment, saturating at a specified threshold value. We specified a Global contract for Incr_Until_Threshold.

Listing 1820: show_increments.ads

```

1 package Show_Increments is
2
3   type Array_Of_Positives is array (Natural range <>) of Positive;
4
5   Increment : constant Natural := 10;
6
7   procedure Incr_Step_Function (A : in out Array_Of_Positives);
8
9 end Show_Increments;
```

Listing 1821: show_increments.adb

```

1 package body Show_Increments is
2
3   procedure Incr_Step_Function (A : in out Array_Of_Positives) is
4
5     Threshold : Positive := Positive'Last;
6
7     procedure Incr_Until_Threshold (I : Integer) with
8       Global => (Input => Threshold,
9                 In_Out => A);
10
11     procedure Incr_Until_Threshold (I : Integer) is
12     begin
13       if Threshold - Increment <= A (I) then
14         A (I) := Threshold;
```

(continues on next page)

(continued from previous page)

```

15     else
16         A (I) := A (I) + Increment;
17     end if;
18 end Incr_Until_Threshold;
19
20 begin
21     for I in A'Range loop
22         if I > A'First then
23             Threshold := A (I - 1);
24         end if;
25         Incr_Until_Threshold (I);
26     end loop;
27 end Incr_Step_Function;
28
29 end Show_Increments;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_07
MD5: 8e28a005cd9d78947e4bfc60db708bf5

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_increments.adb:8:09: info: data dependencies proved

Everything is fine here. Specifically, the Global contract is correct. It mentions both Threshold, which is read but not written in the procedure, and A, which is both read and written. The fact that A is a parameter of an enclosing unit doesn't prevent us from using it inside the Global contract; it really is global to Incr_Until_Threshold. We didn't mention Increment since it's a static constant.

60.5.8 Example #8

We now go back to the procedure Test_Index from [Example #4](#) (page 1446) and correct the missing initializations. We want to know if the Global contract of Test_Index is correct.

Listing 1822: show_biggest_increasing_sequence.ads

```

1 package Show_Biggest_Increasing_Sequence is
2
3     type Array_Of_Positives is array (Integer range <>) of Positive;
4
5     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
6         return Natural;
7
8 end Show_Biggest_Increasing_Sequence;

```

Listing 1823: show_biggest_increasing_sequence.adb

```

1 package body Show_Biggest_Increasing_Sequence is
2
3     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
4         return Natural
5     is
6         Max      : Natural := 0;
7         End_Of_Seq : Boolean;

```

(continues on next page)

(continued from previous page)

```

8      Size_Of_Seq : Natural := 0;
9      Beginning   : Integer := A'First - 1;
10
11     procedure Test_Index (Current_Index : Integer) with
12       Global => (In_Out => (Beginning, Max, Size_Of_Seq),
13         Output => End_Of_Seq,
14         Input  => Current_Index)
15     is
16     begin
17       if A (Current_Index) >= Max then
18         Max := A (Current_Index);
19         End_Of_Seq := False;
20       else
21         Max      := 0;
22         End_Of_Seq := True;
23         Size_Of_Seq := Current_Index - Beginning;
24         Beginning  := Current_Index;
25       end if;
26     end Test_Index;
27
28     Biggest_Seq : Natural := 0;
29
30     begin
31       for I in A'Range loop
32         Test_Index (I);
33         if End_Of_Seq then
34           Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
35         end if;
36       end loop;
37       return Biggest_Seq;
38     end Size_Of_Biggest_Increasing_Sequence;
39
40 end Show_Biggest_Increasing_Sequence;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_08
MD5: 86fb934c32a38f6841ef736780b2e3b2

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
show_biggest_increasing_sequence.adb:14:30: error: global item cannot reference_
↳ parameter of subprogram "Test_Index"
gnatprove: error during generation of Global contracts

```

The contract in this example is not correct: `Current_Index` is a parameter of `Test_Index`, so we shouldn't reference it as a global variable. Also, we should have listed variable `A` from the outer scope as an Input in the Global contract.

60.5.9 Example #9

Next, we change the Global contract of `Test_Index` into a Depends contract. In general, we don't need both contracts because the set of global variables accessed can be deduced from the Depends contract.

Listing 1824: show_biggest_increasing_sequence.ads

```

1 package Show_Biggest_Increasing_Sequence is
2
3   type Array_Of_Positives is array (Integer range <>) of Positive;
4
5   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
6     return Natural;
7
8 end Show_Biggest_Increasing_Sequence;
```

Listing 1825: show_biggest_increasing_sequence.adb

```

1 package body Show_Biggest_Increasing_Sequence is
2
3   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
4     return Natural
5   is
6     Max      : Natural := 0;
7     End_Of_Seq : Boolean;
8     Size_Of_Seq : Natural := 0;
9     Beginning  : Integer := A'First - 1;
10
11    procedure Test_Index (Current_Index : Integer) with
12      Depends => ((Max, End_Of_Seq) => (A, Current_Index, Max),
13        (Size_Of_Seq, Beginning) =>
14          + (A, Current_Index, Max, Beginning))
15    is
16    begin
17      if A (Current_Index) >= Max then
18        Max := A (Current_Index);
19        End_Of_Seq := False;
20      else
21        Max      := 0;
22        End_Of_Seq := True;
23        Size_Of_Seq := Current_Index - Beginning;
24        Beginning  := Current_Index;
25      end if;
26    end Test_Index;
27
28    Biggest_Seq : Natural := 0;
29
30  begin
31    for I in A'Range loop
32      Test_Index (I);
33      if End_Of_Seq then
34        Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
35      end if;
36    end loop;
37    return Biggest_Seq;
38  end Size_Of_Biggest_Increasing_Sequence;
39
40 end Show_Biggest_Increasing_Sequence;
```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_09
MD5: d54ac5d4266738b1bf64869131644b33

Prover output


```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:7:07: info: initialization of "End_Of_Seq"
↳proved
show_biggest_increasing_sequence.adb:11:17: info: initialization of "End_Of_Seq"
↳proved
show_biggest_increasing_sequence.adb:12:09: info: flow dependencies proved
show_biggest_increasing_sequence.ads:5:13: info: implicit aspect Always_Terminates
↳on "Size_Of_Biggest_Increasing_Sequence" has been proved, subprogram will
↳terminate

```

This example is correct. Some of the dependencies, such as `Size_Of_Seq` depending on `Beginning`, come directly from the assignments in the subprogram. Since the control flow influences the final value of all of the outputs, the variables that are being read, `A`, `Current_Index`, and `Max`, are present in every dependency relation. Finally, the dependencies of `Size_Of_Seq` and `Beginning` on themselves are because they may not be modified by the subprogram execution.

60.5.10 Example #10

The subprogram `Identity` swaps the value of its parameter two times. Its `Depends` contract says that the final value of `X` only depends on its initial value and likewise for `Y`.

Listing 1826: `show_swap.ads`

```

1 package Show_Swap is
2
3     procedure Swap (X, Y : in out Positive);
4
5     procedure Identity (X, Y : in out Positive) with
6         Depends => (X => X,
7                     Y => Y);
8
9 end Show_Swap;

```

Listing 1827: `show_swap.adb`

```

1 package body Show_Swap is
2
3     procedure Swap (X, Y : in out Positive) is
4         Tmp : constant Positive := X;
5     begin
6         X := Y;
7         Y := Tmp;
8     end Swap;
9
10    procedure Identity (X, Y : in out Positive) is
11    begin
12        Swap (X, Y);
13        Swap (Y, X);
14    end Identity;
15
16 end Show_Swap;

```

Code block metadata

Project: Courses.Intro_To_Spark.Flow_Analysis.Example_10
MD5: 8567ecele5bbc190f62dd483785d078a

Prover output

```
Phase 1 of 3: generation of data representation information ...  
Phase 2 of 3: generation of Global contracts ...  
Phase 3 of 3: analysis of data and information flow ...  
show_swap.ads:6:18: medium: missing dependency "X => Y"  
show_swap.ads:7:18: medium: missing dependency "Y => X"  
gnatprove: unproved check messages considered as errors
```

This code is correct, but flow analysis can't verify the Depends contract of Identity because we didn't supply a Depends contract for Swap. Therefore, flow analysis assumes that all outputs of Swap, X and Y, depend on all its inputs, both X and Y's initial values. To prevent this, we should manually specify a Depends contract for Swap.

PROOF OF PROGRAM INTEGRITY

This section presents the proof capability of GNATprove, a major tool for the SPARK language. We focus here on the simpler proofs that you'll need to write to verify your program's integrity. The primary objective of performing proof of your program's integrity is to ensure the absence of runtime errors during its execution.

The analysis steps discussed here are only sound if you've previously performed *Flow Analysis* (page 1431). You shouldn't proceed further if you still have unjustified flow analysis messages for your program.

61.1 Runtime Errors

There's always the potential for errors that aren't detected during compilation to occur during a program's execution. These errors, called runtime errors, are those targeted by GNATprove.

There are various kinds of runtime errors, the most common being references that are out of the range of an array (*buffer overflow*³⁹⁷ in Ada), subtype range violations, overflows in computations, and divisions by zero. The code below illustrates many examples of possible runtime errors, all within a single statement. Look at the assignment statement setting the $I + J$ 'th cell of an array *A* to the value P / Q .

Listing 1828: show_runtime_errors.ads

```
1 package Show_Runtime_Errors is
2
3     type Nat_Array is array (Integer range <>) of Natural;
4
5     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);
6
7 end Show_Runtime_Errors;
```

Listing 1829: show_runtime_errors.adb

```
1 package body Show_Runtime_Errors is
2
3     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
4     begin
5         A (I + J) := P / Q;
6     end Update;
7
8 end Show_Runtime_Errors;
```

Code block metadata

³⁹⁷ https://en.wikipedia.org/wiki/Buffer_overflow

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Runtime_Errors
MD5: c0718b8cb6138b84a99e0040e2a9164e

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_runtime_errors.adb:5:12: medium: overflow check might fail, cannot prove
↳ lower bound for I + J [reason for check: result of addition must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (if J >= 0 then I <=
↳ Integer'Last - J else I >= Integer'First - J) to subprogram at show_runtime_
↳ errors.ads:5]
show_runtime_errors.adb:5:12: medium: array index check might fail [reason for
↳ check: result of addition must be a valid index into the array] [possible fix:
↳ add precondition (if J >= 0 then I <= A'Last - J else I >= A'First - J) to
↳ subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: divide by zero might fail [possible fix: add
↳ precondition (Q /= 0) to subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: overflow check might fail, cannot prove
↳ lower bound for P / Q [reason for check: result of division must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (P / Q >= Integer'First)
↳ to subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: range check might fail, cannot prove lower
↳ bound for P / Q [reason for check: result of division must fit in the target
↳ type of the assignment] [possible fix: add precondition (P / Q >= Natural'First)
↳ to subprogram at show_runtime_errors.ads:5]
gnatprove: unproved check messages considered as errors
```

There are quite a number of errors that may occur when executing this code. If we don't know anything about the values of I, J, P, and Q, we can't rule out any of those errors.

First, the computation of $I + J$ can overflow, for example if I is `Integer'Last` and J is positive.

```
A (Integer'Last + 1) := P / Q;
```

Next, the sum, which is used as an array index, may not be in the range of the index of the array.

```
A (A'Last + 1) := P / Q;
```

On the other side of the assignment, the division may also overflow, though only in the very special case where P is `Integer'First` and Q is -1 because of the asymmetric range of signed integer types.

```
A (I + J) := Integer'First / -1;
```

The division is also not allowed if Q is 0.

```
A (I + J) := P / 0;
```

Finally, since the array contains natural numbers, it's also an error to store a negative value in it.

```
A (I + J) := 1 / -1;
```

The compiler generates checks in the executable code corresponding to each of those runtime errors. Each check raises an exception if it fails. For the above assignment statement, we can see examples of exceptions raised due to failed checks for each of the different cases above.

```

A (Integer'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
-- raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
-- raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
-- raised CONSTRAINT_ERROR : divide by zero

```

These runtime checks are costly, both in terms of program size and execution time. It may be appropriate to remove them if we can statically ensure they aren't needed at runtime, in other words if we can prove that the condition tested for can never occur.

This is where the analysis done by GNATprove comes in. It can be used to demonstrate statically that none of these errors can ever occur at runtime. Specifically, GNATprove logically interprets the meaning of every instruction in the program. Using this interpretation, GNATprove generates a logical formula called a *verification condition* for each check that would otherwise be required by the Ada (and hence SPARK) language.

```

A (Integer'Last + 1) := P / Q;
-- medium: overflow check might fail

A (A'Last + 1) := P / Q;
-- medium: array index check might fail

A (I + J) := Integer'First / (-1);
-- medium: overflow check might fail

A (I + J) := 1 / (-1);
-- medium: range check might fail

A (I + J) := P / 0;
-- medium: divide by zero might fail

```

GNATprove then passes these verification conditions to an automatic prover, stated as conditions that must be true to avoid the error. If every such condition can be validated by a prover (meaning that it can be mathematically shown to always be true), we've been able to prove that no error can ever be raised at runtime when executing that program.

61.2 Modularity

To scale to large programs, GNATprove performs proofs on a per-subprogram basis by relying on preconditions and postconditions to properly summarize the input and output state of each subprogram. More precisely, when verifying the body of a subprogram, GNATprove assumes it knows nothing about the possible initial values of its parameters and of the global variables it accesses except what you state in the subprogram's precondition. If you don't specify a precondition, it can't make any assumptions.

For example, the following code shows that the body of `Increment` can be successfully verified: its precondition constrains the value of its parameter `X` to be less than `Integer'Last` so we know the overflow check is always false.

In the same way, when a subprogram is called, GNATprove assumes its `out` and `in out` parameters and the global variables it writes can be modified in any way compatible with their

postconditions. For example, since `Increment` has no postcondition, GNATprove doesn't know that the value of `X` after the call is always less than `Integer'Last`. Therefore, it can't prove that the addition following the call to `Increment` can't overflow.

Listing 1830: `show_modularity.adb`

```
1 procedure Show_Modularity is
2
3     procedure Increment (X : in out Integer) with
4       Pre => X < Integer'Last is
5     begin
6       X := X + 1;
7       -- info: overflow check proved
8     end Increment;
9
10    X : Integer;
11  begin
12    X := Integer'Last - 2;
13    Increment (X);
14    -- After the call, GNATprove no longer knows the value of X
15
16    X := X + 1;
17    -- medium: overflow check might fail
18  end Show_Modularity;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Modularity_1
MD5: ca8ff8d29792fd5a06f7cb0158e13689

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_modularity.adb:6:14: info: overflow check proved
show_modularity.adb:10:04: info: initialization of "X" proved
show_modularity.adb:13:04: info: precondition proved
show_modularity.adb:16:11: medium: overflow check might fail, cannot prove upper_
↳ bound for X + 1 [reason for check: result of addition must fit in a 32-bits_
↳ machine integer] [possible fix: call at line 13 should mention X (for argument_
↳ X) in a postcondition]
gnatprove: unproved check messages considered as errors
```

61.2.1 Exceptions

There are two cases where GNATprove doesn't require modularity and hence doesn't make the above assumptions. First, local subprograms without contracts can be inlined if they're simple enough and are neither recursive nor have multiple return points. If we remove the contract from `Increment`, it fits the criteria for inlining.

Listing 1831: `show_modularity.adb`

```
1 procedure Show_Modularity is
2
3     procedure Increment (X : in out Integer) is
4     begin
5       X := X + 1;
6       -- info: overflow check proved, in call inlined at...
7     end Increment;
8
```

(continues on next page)

(continued from previous page)

```

9      X : Integer;
10  begin
11      X := Integer'Last - 2;
12      Increment (X);
13      X := X + 1;
14      -- info: overflow check proved
15  end Show_Modularity;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Modularity_2
 MD5: 448d576897c3e4606cd4b90621aad63a

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_modularity.adb:5:14: info: overflow check proved, in call inlined at show_
↳modularity.adb:12
show_modularity.adb:9:04: info: initialization of "X" proved
show_modularity.adb:13:11: info: overflow check proved

```

GNATprove now sees the call to Increment exactly as if the increment on X was done outside that call, so it can successfully verify that neither addition can overflow.

Note

For more details on contextual analysis of subprograms, see the [SPARK User's Guide](#)³⁹⁸.

The other case involves functions. If we define a function as an expression function, with or without contracts, GNATprove uses the expression itself as the postcondition on the result of the function.

In our example, replacing Increment with an expression function allows GNATprove to successfully verify the overflow check in the addition.

Listing 1832: show_modularity.adb

```

1  procedure Show_Modularity is
2
3      function Increment (X : Integer) return Integer is
4          (X + 1)
5          -- info: overflow check proved
6          with Pre => X < Integer'Last;
7
8      X : Integer;
9  begin
10     X := Integer'Last - 2;
11     X := Increment (X);
12     X := X + 1;
13     -- info: overflow check proved
14  end Show_Modularity;

```

Code block metadata

³⁹⁸ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_subprogram_contracts.html#contextual-analysis-of-subprograms-without-contracts

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Modularity_3
MD5: b2b67845362929472e4e23867fcbd5e7

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_modularity.adb:3:13: info: implicit aspect Always_Terminates on "Increment"
↳ has been proved, subprogram will terminate
show_modularity.adb:4:09: info: overflow check proved
show_modularity.adb:8:04: info: initialization of "X" proved
show_modularity.adb:11:09: info: precondition proved
show_modularity.adb:12:11: info: overflow check proved
```

Note

For more details on expression functions, see the [SPARK User's Guide](#)³⁹⁹.

61.3 Contracts

Ada contracts are perfectly suited for formal verification, but are primarily designed to be checked at runtime. When you specify the `-gnata` switch, the compiler generates code that verifies the contracts at runtime. If an Ada contract isn't satisfied for a given subprogram call, the program raises the `Assert_Failure` exception. This switch is particularly useful during development and testing, but you may also retain run-time execution of assertions, and specifically preconditions, during the program's deployment to avoid an inconsistent state.

Consider the incorrect call to `Increment` below, which violates its precondition. One way to detect this error is by compiling the function with assertions enabled and testing it with inputs that trigger the violation. Another way, one that doesn't require guessing the needed inputs, is to run GNATprove.

Listing 1833: `show_precondition_violation.adb`

```
1 procedure Show_Precondition_Violation is
2
3   procedure Increment (X : in out Integer) with
4     Pre => X < Integer'Last is
5   begin
6     X := X + 1;
7   end Increment;
8
9   X : Integer;
10
11 begin
12   X := Integer'Last;
13   Increment (X);
14 end Show_Precondition_Violation;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Precondition_Violation
MD5: 60cb889128fc6bca10e21b1baf041258

³⁹⁹ https://docs.adacore.com/live/wave/spark2014/html/spark2014 Ug/en/source/specification_features.html#expression-functions

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_precondition_violation.adb:13:04: medium: precondition might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : failed precondition from show_precondition_
↳ violation.adb:4
```

Similarly, consider the incorrect implementation of function `Absolute` below, which violates its postcondition. Likewise, one way to detect this error is by compiling the function with assertions enabled and testing with inputs that trigger the violation. Another way, one which again doesn't require finding the inputs needed to demonstrate the error, is to run GNATprove.

Listing 1834: `show_postcondition_violation.adb`

```
1 procedure Show_Postcondition_Violation is
2
3   procedure Absolute (X : in out Integer) with
4     Post => X >= 0 is
5   begin
6     if X > 0 then
7       X := -X;
8     end if;
9   end Absolute;
10
11   X : Integer;
12
13 begin
14   X := 1;
15   Absolute (X);
16 end Show_Postcondition_Violation;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Postcondition_Violation
MD5: fb1340de7e082d801f177bd8a0cf90a6
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_postcondition_violation.adb:4:14: medium: postcondition might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : failed postcondition from show_
↳ postcondition_violation.adb:4
```

The benefits of dynamically checking contracts extends beyond making testing easier. Early failure detection also allows an easier recovery and facilitates debugging, so you may want to enable these checks at runtime to terminate execution before some damaging or hard-to-debug action occurs.

GNATprove statically analyses preconditions and postconditions. It verifies preconditions every time a subprogram is called, which is the runtime semantics of contracts. Postconditions, on the other hand, are verified once as part of the verification of the subprogram's body. For example, GNATprove must wait until `Increment` is improperly called to detect the precondition violation, since a precondition is really a contract for the caller. On the other hand, it doesn't need `Absolute` to be called to detect that its postcondition doesn't hold for all its possible inputs.

Note

For more details on pre and postconditions, see the [SPARK User's Guide](#)⁴⁰⁰.

61.3.1 Executable Semantics

Expressions in Ada contracts have the same semantics as Boolean expressions elsewhere, so runtime errors can occur during their computation. To simplify both debugging of assertions and combining testing and static verification, the same semantics are used by GNATprove.

While proving programs, GNATprove verifies that no error can ever be raised during the execution of the contracts. However, you may sometimes find those semantics too heavy, in particular with respect to overflow checks, because they can make it harder to specify an appropriate precondition. We see this in the function `Add` below.

Listing 1835: `show_executable_semantics.adb`

```
1 procedure Show_Executable_Semantics
2   with SPARK_Mode => On
3 is
4   function Add (X, Y : Integer) return Integer is (X + Y)
5     with Pre => X + Y in Integer;
6
7   X : Integer;
8 begin
9   X := Add (Integer'Last, 1);
10 end Show_Executable_Semantics;
```

Code block metadata

Project: `Courses.Intro_To_Spark.Proof_of_Program_Integrity.Executable_Semantics`
MD5: `d85fa0507d7c35fb98ade7815020117e`

Build output

```
show_executable_semantics.adb:5:24: warning: explicit membership test may be
↳ optimized away [enabled by default]
show_executable_semantics.adb:5:24: warning: use 'Valid attribute instead [enabled
↳ by default]
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_executable_semantics.adb:5:20: medium: overflow check might fail, cannot
↳ prove lower bound for X + Y [reason for check: result of addition must fit in a
↳ 32-bits machine integer] [possible fix: use pragma Overflow_Mode or switch -
(continues on next page)
```

⁴⁰⁰ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#preconditions

(continued from previous page)

```

↳gnato13 or unit SPARK.Big_Integers]
show_executable_semantics.adb:9:09: medium: precondition might fail, cannot prove_
↳upper bound for Add (Integer'Last, 1)
gnatprove: unproved check messages considered as errors

```

Runtime output

```
raised CONSTRAINT_ERROR : show_executable_semantics.adb:5 overflow check failed
```

GNATprove issues a message on this code warning about a possible overflow when computing the sum of X and Y in the precondition. Indeed, since expressions in assertions have normal Ada semantics, this addition can overflow, as you can easily see by compiling and running the code that calls Add with arguments `Integer'Last` and 1.

On the other hand, you sometimes may prefer GNATprove to use the mathematical semantics of addition in contracts while the generated code still properly verifies that no error is ever raised at runtime in the body of the program. You can get this behavior by using the compiler switch `-gnato??` (for example `-gnato13`), which allows you to independently set the overflow mode in code (the first digit) and assertions (the second digit). For both, you can either reduce the number of overflow checks (the value 2), completely eliminate them (the value 3), or preserve the default Ada semantics (the value 1).

Note

For more details on overflow modes, see the [SPARK User's Guide](#)⁴⁰¹.

61.3.2 Additional Assertions and Contracts

As we've seen, a key feature of SPARK is that it allows us to state properties to check using assertions and contracts. SPARK supports preconditions and postconditions as well as assertions introduced by the Assert pragma.

The SPARK language also includes new contract types used to assist formal verification. The new pragma Assume is treated as an assertion during execution but introduces an assumption when proving programs. Its value is a Boolean expression which GNATprove assumes to be true without any attempt to verify that it's true. You'll find this feature useful, but you must use it with great care. Here's an example of using it.

Listing 1836: incr.adb

```

1 procedure Incr (X : in out Integer) is
2   begin
3     pragma Assume (X < Integer'Last);
4     X := X + 1;
5   end Incr;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Pragma_Assume
MD5: bfbcb8aca259d7516b6acae571f8c2

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...

```

(continues on next page)

⁴⁰¹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/overflow_modes.html

(continued from previous page)

Phase 3 of 3: flow analysis and proof ...
 incr.adb:4:11: info: overflow check proved

Note

For more details on pragma Assume, see the [SPARK User's Guide](#)⁴⁰².

The `Contract_Cases` aspect is another construct introduced for GNATprove, but which also acts as an assertion during execution. It allows you to specify the behavior of a subprogram using a disjunction of cases. Each element of a `Contract_Cases` aspect is a *guard*, which is evaluated before the call and may only reference the subprogram's inputs, and a *consequence*. At each call of the subprogram, one and only one guard is permitted to evaluate to **True**. The consequence of that case is a contract that's required to be satisfied when the subprogram returns.

Listing 1837: absolute.adb

```

1 procedure Absolute (X : in out Integer) with
2   Pre           => X > Integer'First,
3   Contract_Cases => (X < 0 => X = -X'Old,
4                     X >= 0 => X = X'Old)
5 is
6 begin
7   if X < 0 then
8     X := -X;
9   end if;
10 end Absolute;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Absolute
 MD5: 5ac868f35be18bb6fffe2444ecbea28d

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: flow analysis and proof ...
 absolute.adb:3:03: info: disjoint contract or exit cases proved
 absolute.adb:3:03: info: complete contract cases proved
 absolute.adb:3:22: info: contract case proved
 absolute.adb:3:36: info: overflow check proved
 absolute.adb:4:22: info: contract case proved
 absolute.adb:8:12: info: overflow check proved

Similarly to how it analyzes a subprogram's precondition, GNATprove verifies the `Contract_Cases` only once. It verifies the validity of each consequence (given the truth of its guard) and the disjointness and completeness of the guard conditions (meaning that exactly one guard must be true for each possible set of input values).

Note

For more details on `Contract_Cases`, see the [SPARK User's Guide](#)⁴⁰³.

⁴⁰² https://docs.adacore.com/live/wave/spark2014/html/spark2014 Ug/en/source/assertion_pragmas.html#pragma-assume

⁴⁰³ https://docs.adacore.com/live/wave/spark2014/html/spark2014 Ug/en/source/subprogram_contracts.html#contract-cases

61.4 Debugging Failed Proof Attempts

GNATprove may report an error while verifying a program for any of the following reasons:

- there might be an error in the program; or
- the property may not be provable as written because more information is required; or
- the prover used by GNATprove may be unable to prove a perfectly valid property.

We spend the remainder of this section discussing the sometimes tricky task of debugging failed proof attempts.

61.4.1 Debugging Errors in Code or Specification

First, let's discuss the case where there's indeed an error in the program. There are two possibilities: the code may be incorrect or, equally likely, the specification may be incorrect. As an example, there's an error in our procedure `Incr_Until` below which makes its `Contract_Cases` unprovable.

Listing 1838: `show_failed_proof_attempt.ads`

```

1 package Show_Failed_Proof_Attempt is
2
3   Incremented : Boolean := False;
4
5   procedure Incr_Until (X : in out Natural) with
6     Contract_Cases =>
7     (Incremented => X > X'Old,
8      others      => X = X'Old);
9
10 end Show_Failed_Proof_Attempt;
```

Listing 1839: `show_failed_proof_attempt.adb`

```

1 package body Show_Failed_Proof_Attempt is
2
3   procedure Incr_Until (X : in out Natural) is
4   begin
5     if X < 1000 then
6       X := X + 1;
7       Incremented := True;
8     else
9       Incremented := False;
10    end if;
11  end Incr_Until;
12
13 end Show_Failed_Proof_Attempt;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Failed_Proof_Attempt_1
MD5: 814636ae9df6f4f66ad69f5099a5729b

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_failed_proof_attempt.ads:7:09: medium: contract case might fail
show_failed_proof_attempt.ads:8:09: medium: contract case might fail
gnatprove: unproved check messages considered as errors
```

Since this is an assertion that can be executed, it may help you find the problem if you run the program with assertions enabled on representative sets of inputs. This allows you to find bugs in both the code and its contracts. In this case, testing `Incr_Until` with an input greater than 1000 raises an exception at runtime.

Listing 1840: `show_failed_proof_attempt.ads`

```
1 package Show_Failed_Proof_Attempt is
2
3   Incremented : Boolean := False;
4
5   procedure Incr_Until (X : in out Natural) with
6     Contract_Cases =>
7     (Incremented => X > X'Old,
8      others      => X = X'Old);
9
10 end Show_Failed_Proof_Attempt;
```

Listing 1841: `show_failed_proof_attempt.adb`

```
1 package body Show_Failed_Proof_Attempt is
2
3   procedure Incr_Until (X : in out Natural) is
4   begin
5     if X < 1000 then
6       X := X + 1;
7       Incremented := True;
8     else
9       Incremented := False;
10    end if;
11  end Incr_Until;
12
13 end Show_Failed_Proof_Attempt;
```

Listing 1842: `main.adb`

```
1 with Show_Failed_Proof_Attempt; use Show_Failed_Proof_Attempt;
2
3 procedure Main is
4   Dummy : Integer;
5 begin
6   Dummy := 0;
7   Incr_Until (Dummy);
8
9   Dummy := 1000;
10  Incr_Until (Dummy);
11 end Main;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Failed_Proof_Attempt_2
MD5: bd87cb0f64a6468eaab3cad1678271db

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_failed_proof_attempt.ads:7:09: medium: contract case might fail
show_failed_proof_attempt.ads:8:09: medium: contract case might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA.Assertions.Assertion_Error : failed contract case at show_failed_proof_
↳ attempt.ads:8
```

The error message shows that the first contract case is failing, which means that `Incremented` is **True**. However, if we print the value of `Incremented` before returning, we see that it's **False**, as expected for the input we provided. The error here is that guards of contract cases are evaluated before the call, so our specification is wrong! To correct this, we should either write `X < 1000` as the guard of the first case or use a standard postcondition with an if-expression.

61.4.2 Debugging Cases where more Information is Required

Even if both the code and the assertions are correct, GNATprove may still report that it can't prove a verification condition for a property. This can happen for two reasons:

- The property may be unprovable because the code is missing some assertion. One category of these cases is due to the modularity of the analysis which, as we discussed above, means that GNATprove only knows about the properties of your subprograms that you have explicitly written.
- There may be some information missing in the logical model of the program used by GNATprove.

Let's look at the case where the code and the specification are correct but there's some information missing. As an example, GNATprove finds the postcondition of `Increase` to be unprovable.

Listing 1843: `show_failed_proof_attempt.ads`

```
1 package Show_Failed_Proof_Attempt is
2
3   C : Natural := 100;
4
5   procedure Increase (X : in out Natural) with
6     Post => (if X'Old < C then X > X'Old else X = C);
7
8 end Show_Failed_Proof_Attempt;
```

Listing 1844: `show_failed_proof_attempt.adb`

```
1 package body Show_Failed_Proof_Attempt is
2
3   procedure Increase (X : in out Natural) is
4   begin
5     if X < 90 then
6       X := X + 10;
7     elsif X >= C then
8       X := C;
9     else
10      X := X + 1;
11    end if;
12  end Increase;
13
14 end Show_Failed_Proof_Attempt;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Failed_Proof_Attempt_3
MD5: e01fc27a981bcb80757f30c94768237e
```


Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_failed_proof_attempt.ads:6:49: medium: postcondition might fail, cannot prove_
  X = C
gnatprove: unproved check messages considered as errors
```

This postcondition is a conditional. It says that if the parameter (X) is less than a certain value (C), its value will be increased by the procedure while if it's greater, its value will be set to C (saturated). When C has the value 100, the code of `Increase` adds 10 to the value of X if it was initially less than 90, increments X by 1 if it was between 90 and 99, and sets X to 100 if it was greater or equal to 100. This behavior does satisfy the postcondition, so why is the postcondition not provable?

The values in the counterexample returned by GNATprove in its message gives us a clue: `C = 0 and X = 10 and X'Old = 0`. Indeed, if C is not equal to 100, our reasoning above is incorrect: the values of 0 for C and X on entry indeed result in X being 10 on exit, which violates the postcondition!

We probably didn't expect the value of C to change, or at least not to go below 90. But, in that case, we should have stated so by either declaring C to be constant or by adding a precondition to the `Increase` subprogram. If we do either of those, GNATprove is able to prove the postcondition.

61.4.3 Debugging Prover Limitations

Finally, there are cases where GNATprove provides a perfectly valid verification condition for a property, but it's nevertheless not proved by the automatic prover that runs in the later stages of the tool's execution. This is quite common. Indeed, GNATprove produces its verification conditions in first-order logic, which is not decidable, especially in combination with the rules of arithmetic. Sometimes, the automatic prover just needs more time. Other times, the prover will abandon the search almost immediately or loop forever without reaching a conclusive answer (either a proof or a counterexample).

For example, the postcondition of our GCD function below — which calculates the value of the GCD of two positive numbers using Euclidean's algorithm — can't be verified with GNATprove's default settings.

Listing 1845: `show_failed_proof_attempt.ads`

```
1 package Show_Failed_Proof_Attempt is
2
3   function GCD (A, B : Positive) return Positive with
4     Post =>
5       A mod GCD'Result = 0
6       and B mod GCD'Result = 0;
7
8 end Show_Failed_Proof_Attempt;
```

Listing 1846: `show_failed_proof_attempt.adb`

```
1 package body Show_Failed_Proof_Attempt is
2
3   function GCD (A, B : Positive) return Positive is
4   begin
5     if A > B then
6       return GCD (A - B, B);
7     elsif B > A then
8       return GCD (A, B - A);
```

(continues on next page)

(continued from previous page)

```

9      else
10         return A;
11     end if;
12 end GCD;
13
14 end Show_Failed_Proof_Attempt;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Failed_Proof_Attempt_4
MD5: a6f1a39ceb0793df8a00691d59a5d9ce

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_failed_proof_attempt.adb:6:10: medium: implicit aspect Always_Terminates on
↳ "GCD" could be incorrect, subprogram is recursive [possible fix: annotate "GCD"
↳ with a Subprogram_Variant aspect]
show_failed_proof_attempt.adb:8:10: medium: implicit aspect Always_Terminates on
↳ "GCD" could be incorrect, subprogram is recursive [possible fix: annotate "GCD"
↳ with a Subprogram_Variant aspect]
show_failed_proof_attempt.ads:5:08: medium: postcondition might fail, cannot prove
↳ A mod GCD'Result = 0
gnatprove: unproved check messages considered as errors
```

The first thing we try is increasing the amount of time the prover is allowed to spend on each verification condition using the `--timeout` option of GNATprove (e.g., by using the dialog box in GNAT Studio). In this example, increasing it to one minute, which is relatively high, doesn't help. We can also specify an alternative automatic prover — if we have one — using the option `--prover` of GNATprove (or the dialog box). For our postcondition, we tried Alt-Ergo, cvc5, and Z3 without any luck.

Listing 1847: show_failed_proof_attempt.ads

```

1 package Show_Failed_Proof_Attempt is
2
3     function GCD (A, B : Positive) return Positive with
4         Post =>
5             A mod GCD'Result = 0
6             and B mod GCD'Result = 0;
7
8 end Show_Failed_Proof_Attempt;
```

Listing 1848: show_failed_proof_attempt.adb

```

1 package body Show_Failed_Proof_Attempt is
2
3     function GCD (A, B : Positive) return Positive
4     is
5         Result : Positive;
6     begin
7         if A > B then
8             Result := GCD (A - B, B);
9             pragma Assert ((A - B) mod Result = 0);
10            -- info: assertion proved
11            pragma Assert (B mod Result = 0);
12            -- info: assertion proved
13            pragma Assert (A mod Result = 0);
```

(continues on next page)

(continued from previous page)

```

14      -- medium: assertion might fail
15      elsif B > A then
16          Result := GCD (A, B - A);
17          pragma Assert ((B - A) mod Result = 0);
18          -- info: assertion proved
19      else
20          Result := A;
21      end if;
22      return Result;
23  end GCD;
24
25 end Show_Failed_Proof_Attempt;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Failed_Proof_Attempt_5
 MD5: 954ecbf2177705770c3a44a477c1de17

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_failed_proof_attempt.adb:5:07: info: initialization of "Result" proved
show_failed_proof_attempt.adb:8:17: medium: implicit aspect Always_Terminates on
↳ "GCD" could be incorrect, subprogram is recursive [possible fix: annotate "GCD"
↳ with a Subprogram_Variant aspect]
show_failed_proof_attempt.adb:8:27: info: range check proved
show_failed_proof_attempt.adb:9:25: info: assertion proved
show_failed_proof_attempt.adb:9:33: info: division check proved
show_failed_proof_attempt.adb:11:25: info: assertion proved
show_failed_proof_attempt.adb:11:27: info: division check proved
show_failed_proof_attempt.adb:13:25: medium: assertion might fail [possible fix:
↳ subprogram at show_failed_proof_attempt.ads:3 should mention A in a precondition]
show_failed_proof_attempt.adb:13:27: info: division check proved
show_failed_proof_attempt.adb:16:17: medium: implicit aspect Always_Terminates on
↳ "GCD" could be incorrect, subprogram is recursive [possible fix: annotate "GCD"
↳ with a Subprogram_Variant aspect]
show_failed_proof_attempt.adb:16:30: info: range check proved
show_failed_proof_attempt.adb:17:25: info: assertion proved
show_failed_proof_attempt.adb:17:33: info: division check proved
show_failed_proof_attempt.ads:5:10: info: division check proved
show_failed_proof_attempt.ads:6:12: medium: postcondition might fail, cannot prove
↳ B mod GCD'Result = 0
show_failed_proof_attempt.ads:6:14: info: division check proved
gnatprove: unproved check messages considered as errors

```

To better understand the reason for the failure, we added intermediate assertions to simplify the proof and pin down the part that's causing the problem. Adding such assertions is often a good idea when trying to understand why a property is not proved. Here, provers can't verify that if both $A - B$ and B can be divided by Result so can A . This may seem surprising, but non-linear arithmetic, involving, for example, multiplication, modulo, or exponentiation, is a difficult topic for provers and is not handled very well in practice by any of the general-purpose ones like Alt-Ergo, cvc5, or Z3.

Note

For more details on how to investigate unproved checks, see the [SPARK User's Guide](https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_investigate_unproved_)⁴⁰⁴.

⁴⁰⁴ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_investigate_unproved_

61.5 Code Examples / Pitfalls

We end with some code examples and pitfalls.

61.5.1 Example #1

The package `Lists` defines a linked-list data structure. We call `Link(I,J)` to make a link from index `I` to index `J` and call `Goes_To(I,J)` to determine if we've created a link from index `I` to index `J`. The postcondition of `Link` uses `Goes_To` to state that there must be a link between its arguments once `Link` completes.

Listing 1849: `lists.ads`

```

1 package Lists with SPARK_Mode is
2
3   type Index is new Integer;
4
5   function Goes_To (I, J : Index) return Boolean;
6
7   procedure Link (I, J : Index) with Post => Goes_To (I, J);
8
9 private
10
11   type Cell (Is_Set : Boolean := True) is record
12     case Is_Set is
13       when True =>
14         Next : Index;
15       when False =>
16         null;
17     end case;
18   end record;
19
20   type Cell_Array is array (Index) of Cell;
21
22   Memory : Cell_Array;
23
24 end Lists;
```

Listing 1850: `lists.adb`

```

1 package body Lists with SPARK_Mode is
2
3   function Goes_To (I, J : Index) return Boolean is
4   begin
5     if Memory (I).Is_Set then
6       return Memory (I).Next = J;
7     end if;
8     return False;
9   end Goes_To;
10
11   procedure Link (I, J : Index) is
12   begin
13     Memory (I) := (Is_Set => True, Next => J);
14   end Link;
15
16 end Lists;
```

Code block metadata

checks.html

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_01
MD5: c2246948c584304d5694b49b4d1fd0fc

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
lists.ads:7:47: medium: postcondition might fail [possible fix: you should
↳ consider adding a postcondition to function Goes_To or turning it into an
↳ expression function]
gnatprove: unproved check messages considered as errors

This example is correct, but can't be verified by GNATprove. This is because Goes_To itself has no postcondition, so nothing is known about its result.

61.5.2 Example #2

We now redefine Goes_To as an expression function.

Listing 1851: lists.ads

```
1 package Lists with SPARK_Mode is
2
3   type Index is new Integer;
4
5   function Goes_To (I, J : Index) return Boolean;
6
7   procedure Link (I, J : Index) with Post => Goes_To (I, J);
8
9 private
10
11   type Cell (Is_Set : Boolean := True) is record
12     case Is_Set is
13       when True =>
14         Next : Index;
15       when False =>
16         null;
17     end case;
18   end record;
19
20   type Cell_Array is array (Index) of Cell;
21
22   Memory : Cell_Array;
23
24   function Goes_To (I, J : Index) return Boolean is
25     (Memory (I).Is_Set and then Memory (I).Next = J);
26
27 end Lists;
```

Listing 1852: lists.adb

```
1 package body Lists with SPARK_Mode is
2
3   procedure Link (I, J : Index) is
4   begin
5     Memory (I) := (Is_Set => True, Next => J);
6   end Link;
7
8 end Lists;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_02
 MD5: c65953bbe8a5f9fb77a4d94e2dd875f9

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: flow analysis and proof ...
 lists.adb:5:18: info: discriminant check proved
 lists.ads:5:13: info: implicit aspect Always_Terminates on "Goes_To" has been _
 → proved, subprogram will terminate
 lists.ads:7:47: info: postcondition proved
 lists.ads:25:44: info: discriminant check proved

GNATprove can fully prove this version: Goes_To is an expression function, so its body is available for proof (specifically, for creating the postcondition needed for the proof).

61.5.3 Example #3

The package Stacks defines an abstract stack type with a Push procedure that adds an element at the top of the stack and a function Peek that returns the content of the element at the top of the stack (without removing it).

Listing 1853: stacks.ads

```

1 package Stacks with SPARK_Mode is
2
3   type Stack is private;
4
5   function Peek (S : Stack) return Natural;
6   procedure Push (S : in out Stack; E : Natural) with
7     Post => Peek (S) = E;
8
9 private
10
11   Max : constant := 10;
12
13   type Stack_Array is array (1 .. Max) of Natural;
14
15   type Stack is record
16     Top      : Positive;
17     Content  : Stack_Array;
18   end record;
19
20   function Peek (S : Stack) return Natural is
21     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
22
23 end Stacks;
```

Listing 1854: stacks.adb

```

1 package body Stacks with SPARK_Mode is
2
3   procedure Push (S : in out Stack; E : Natural) is
4   begin
5     if S.Top >= Max then
6       return;
7     end if;
8
```

(continues on next page)

(continued from previous page)

```
9      S.Top := S.Top + 1;  
10     S.Content (S.Top) := E;  
11   end Push;  
12  
13 end Stacks;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_03
MD5: 917d624916c5ef14c4e454d6c56414fd

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
stacks.ads:7:14: medium: postcondition might fail
gnatprove: unproved check messages considered as errors

This example isn't correct. The postcondition of Push is only satisfied if the stack isn't full when we call Push.

61.5.4 Example #4

We now change the behavior of Push so it raises an exception when the stack is full instead of returning.

Listing 1855: stacks.ads

```
1 package Stacks with SPARK_Mode is  
2  
3   type Stack is private;  
4  
5   Is_Full_E : exception;  
6  
7   function Peek (S : Stack) return Natural;  
8   procedure Push (S : in out Stack; E : Natural) with  
9     Post => Peek (S) = E;  
10  
11 private  
12  
13   Max : constant := 10;  
14  
15   type Stack_Array is array (1 .. Max) of Natural;  
16  
17   type Stack is record  
18     Top      : Positive;  
19     Content : Stack_Array;  
20   end record;  
21  
22   function Peek (S : Stack) return Natural is  
23     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);  
24  
25 end Stacks;
```

Listing 1856: stacks.adb

```
1 package body Stacks with SPARK_Mode is  
2  
3   procedure Push (S : in out Stack; E : Natural) is
```

(continues on next page)

(continued from previous page)

```

4   begin
5       if S.Top >= Max then
6           raise Is_Full_E;
7       end if;
8
9       S.Top := S.Top + 1;
10      S.Content (S.Top) := E;
11  end Push;
12
13 end Stacks;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_04
MD5: b573ebe93f85ea171166b6953cbb8956

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
stacks.adb:6:10: medium: unexpected exception might be raised
gnatprove: unproved check messages considered as errors

The postcondition of Push is now proved because GNATprove only considers execution paths leading to normal termination. But it issues a message warning that exception Is_Full_E may be raised at runtime.

61.5.5 Example #5

Let's add a precondition to Push stating that the stack shouldn't be full.

Listing 1857: stacks.ads

```

1  package Stacks with SPARK_Mode is
2
3      type Stack is private;
4
5      Is_Full_E : exception;
6
7      function Peek (S : Stack) return Natural;
8      function Is_Full (S : Stack) return Boolean;
9      procedure Push (S : in out Stack; E : Natural) with
10         Pre  => not Is_Full (S),
11         Post => Peek (S) = E;
12
13  private
14
15      Max : constant := 10;
16
17      type Stack_Array is array (1 .. Max) of Natural;
18
19      type Stack is record
20          Top      : Positive;
21          Content : Stack_Array;
22      end record;
23
24      function Peek (S : Stack) return Natural is
25          (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
26      function Is_Full (S : Stack) return Boolean is (S.Top >= Max);

```

(continues on next page)

(continued from previous page)

```

27
28 end Stacks;

```

Listing 1858: stacks.adb

```

1 package body Stacks with SPARK_Mode is
2
3   procedure Push (S : in out Stack; E : Natural) is
4   begin
5     if S.Top >= Max then
6       raise Is_Full_E;
7     end if;
8     S.Top := S.Top + 1;
9     S.Content (S.Top) := E;
10  end Push;
11
12 end Stacks;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_05
MD5: 63c2dfd68dd5acd91d8d497206e7423e

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
stacks.adb:6:10: info: only expected exception raised
stacks.adb:8:22: info: overflow check proved
stacks.adb:9:19: info: index check proved
stacks.ads:7:14: info: implicit aspect Always_Terminates on "Peek" has been proved,
↳ subprogram will terminate
stacks.ads:8:14: info: implicit aspect Always_Terminates on "Is_Full" has been
↳ proved, subprogram will terminate
stacks.ads:11:14: info: postcondition proved
stacks.ads:25:52: info: index check proved

```

This example is correct. With the addition of the precondition, GNATprove can now verify that Is_Full_E can never be raised at runtime.

61.5.6 Example #6

The package Memories defines a type Chunk that models chunks of memory. Each element of the array, represented by its index, corresponds to one data element. The procedure Read_Record reads two pieces of data starting at index From out of the chunk represented by the value of Memory.

Listing 1859: memories.ads

```

1 package Memories is
2
3   type Chunk is array (Integer range <>) of Integer
4   with Predicate => Chunk'Length >= 10;
5
6   function Is_Too_Coarse (V : Integer) return Boolean;
7
8   procedure Treat_Value (V : out Integer);
9
10 end Memories;

```

Listing 1860: read_record.adb

```

1 with Memories; use Memories;
2
3 procedure Read_Record (Memory : Chunk; From : Integer)
4   with SPARK_Mode => On,
5        Pre => From in Memory'First .. Memory'Last - 2
6 is
7   function Read_One (First : Integer; Offset : Integer) return Integer
8     with Pre => First + Offset in Memory'Range
9   is
10    Value : Integer := Memory (First + Offset);
11    begin
12      if Is_Too_Coarse (Value) then
13        Treat_Value (Value);
14      end if;
15      return Value;
16    end Read_One;
17
18    Data1, Data2 : Integer;
19
20 begin
21   Data1 := Read_One (From, 1);
22   Data2 := Read_One (From, 2);
23 end Read_Record;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_06
MD5: aec8014dc291708999092fa123ee7416

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
read_record.adb:8:24: medium: overflow check might fail, cannot prove lower bound_
↳for First + Offset [reason for check: result of addition must fit in a 32-bits_
↳machine integer] [possible fix: use pragma Overflow_Mode or switch -gnatol3 or_
↳unit SPARK.Big_Integers]
read_record.adb:13:10: warning: no Always_Terminates aspect available for "Treat_
↳Value" [assumed-always-terminates]
read_record.adb:13:10: warning: assuming "Treat_Value" always terminates
gnatprove: unproved check messages considered as errors

```

This example is correct, but it can't be verified by GNATprove, which analyses Read_One on its own and notices that an overflow may occur in its precondition in certain contexts.

61.5.7 Example #7

Let's rewrite the precondition of Read_One to avoid any possible overflow.

Listing 1861: memories.ads

```

1 package Memories is
2
3   type Chunk is array (Integer range <>) of Integer
4   with Predicate => Chunk'Length >= 10;

```

(continues on next page)

(continued from previous page)

```

5
6   function Is_Too_Coarse (V : Integer) return Boolean;
7
8   procedure Treat_Value (V : out Integer);
9
10  end Memories;

```

Listing 1862: read_record.adb

```

1  with Memories; use Memories;
2
3  procedure Read_Record (Memory : Chunk; From : Integer)
4    with SPARK_Mode => On,
5         Pre => From in Memory'First .. Memory'Last - 2
6  is
7    function Read_One (First : Integer; Offset : Integer) return Integer
8      with Pre => First >= Memory'First
9           and then Offset in 0 .. Memory'Last - First
10   is
11     Value : Integer := Memory (First + Offset);
12   begin
13     if Is_Too_Coarse (Value) then
14       Treat_Value (Value);
15     end if;
16     return Value;
17   end Read_One;
18
19   Data1, Data2 : Integer;
20
21 begin
22   Data1 := Read_One (From, 1);
23   Data2 := Read_One (From, 2);
24 end Read_Record;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_07
MD5: 6b4c6a41b652ad76bc7ef8934dcd9bfc

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
read_record.adb:9:49: medium: overflow check might fail, cannot prove lower bound_
↳ for Memory'Last - First [reason for check: result of subtraction must fit in a_
↳ 32-bits machine integer] [possible fix: use pragma Overflow_Mode or switch -
↳ gnato13 or unit SPARK.Big_Integers]
read_record.adb:14:10: warning: no Always_Terminates aspect available for "Treat_
↳ Value" [assumed-always-terminates]
read_record.adb:14:10: warning: assuming "Treat_Value" always terminates
gnatprove: unproved check messages considered as errors

```

This example is also not correct: unfortunately, our attempt to correct Read_One's precondition failed. For example, an overflow will occur at runtime if First is **Integer'Last** and Memory'Last is negative. This is possible here because type Chunk uses **Integer** as base index type instead of **Natural** or **Positive**.

61.5.8 Example #8

Let's completely remove the precondition of Read_One.

Listing 1863: memories.ads

```

1 package Memories is
2
3   type Chunk is array (Integer range <>) of Integer
4     with Predicate => Chunk'Length >= 10;
5
6   function Is_Too_Coarse (V : Integer) return Boolean;
7
8   procedure Treat_Value (V : out Integer);
9
10 end Memories;
```

Listing 1864: read_record.adb

```

1 with Memories; use Memories;
2
3 procedure Read_Record (Memory : Chunk; From : Integer)
4   with SPARK_Mode => On,
5     Pre => From in Memory'First .. Memory'Last - 2
6 is
7   function Read_One (First : Integer; Offset : Integer) return Integer is
8     Value : Integer := Memory (First + Offset);
9   begin
10    if Is_Too_Coarse (Value) then
11      Treat_Value (Value);
12    end if;
13    return Value;
14  end Read_One;
15
16  Data1, Data2 : Integer;
17
18 begin
19   Data1 := Read_One (From, 1);
20   Data2 := Read_One (From, 2);
21 end Read_Record;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_08
MD5: 5a806fb84b50d2dc1f2af428b1bc8d0a

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
read_record.adb:5:51: info: overflow check proved
read_record.adb:7:13: info: implicit aspect Always_Terminates on "Read_One" has
↳ been proved, subprogram will terminate
read_record.adb:8:40: info: overflow check proved, in call inlined at read_record.
↳ adb:19
read_record.adb:8:40: info: index check proved, in call inlined at read_record.
↳ adb:19
read_record.adb:8:40: info: overflow check proved, in call inlined at read_record.
```

(continues on next page)

(continued from previous page)

```

↪adb:20
read_record.adb:8:40: info: index check proved, in call inlined at read_record.
↪adb:20
read_record.adb:11:10: warning: no Always_Terminates aspect available for "Treat_
↪Value" [assumed-always-terminates]
read_record.adb:11:10: warning: assuming "Treat_Value" always terminates

```

This example is correct and fully proved. We could have fixed the contract of `Read_One` to correctly handle both positive and negative values of `Memory'Last`, but we found it simpler to let the function be inlined for proof by removing its precondition.

61.5.9 Example #9

The procedure `Compute` performs various computations on its argument. The computation performed depends on its input range and is reflected in its contract, which we express using a `Contract_Cases` aspect.

Listing 1865: compute.adb

```

1  procedure Compute (X : in out Integer) with
2      Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
3                          (X in 0 .. 199) => X = X'Old + 1,
4                          (X in -199 .. 0) => X = X'Old - 1,
5                          X >= 200 => X = 200,
6                          others => X = -200)
7  is
8  begin
9      if X in -100 .. 100 then
10         X := X * 2;
11     elsif X in 0 .. 199 then
12         X := X + 1;
13     elsif X in -199 .. 0 then
14         X := X - 1;
15     elsif X >= 200 then
16         X := 200;
17     else
18         X := -200;
19     end if;
20 end Compute;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_09
MD5: 51962d1bb6dd1b081ed498dd11559685

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
compute.adb:2:03: medium: contract or exit cases might not be disjoint
compute.adb:3:22: medium: contract case might fail
compute.adb:4:22: medium: contract case might fail
gnatprove: unproved check messages considered as errors

```

This example isn't correct. We duplicated the content of `Compute`'s body in its contract. This is incorrect because the semantics of `Contract_Cases` require disjoint cases, just like a case statement. The counterexample returned by `GNATprove` shows that `X = 0` is covered by two different case-guards (the first and the second).

61.5.10 Example #10

Let's rewrite the contract of Compute to avoid overlapping cases.

Listing 1866: compute.adb

```

1  procedure Compute (X : in out Integer) with
2      Contract_Cases => ((X in 0 .. 199) => X >= X'Old,
3                          (X in -199 .. -1) => X <= X'Old,
4                          X >= 200      => X = 200,
5                          X < -200     => X = -200)
6  is
7  begin
8      if X in -100 .. 100 then
9          X := X * 2;
10         elsif X in 0 .. 199 then
11             X := X + 1;
12         elsif X in -199 .. 0 then
13             X := X - 1;
14         elsif X >= 200 then
15             X := 200;
16         else
17             X := -200;
18         end if;
19     end Compute;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Program_Integrity.Example_10
MD5: 01d33b10fd60f384ffa4ae8feale7d87

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
compute.adb:2:03: medium: contract cases might not be complete
gnatprove: unproved check messages considered as errors

This example is still not correct. GNATprove can successfully prove the different cases are disjoint and also successfully verify each case individually. This isn't enough, though: a Contract_Cases must cover all cases. Here, we forgot the value -200, which is what GNATprove reports in its counterexample.

STATE ABSTRACTION

Abstraction is a key concept in programming that can drastically simplify both the implementation and maintenance of code. It's particularly well suited to SPARK and its modular analysis. This section explains what state abstraction is and how you use it in SPARK. We explain how it impacts GNATprove's analysis both in terms of information flow and proof of program properties.

State abstraction allows us to:

- express dependencies that wouldn't otherwise be expressible because some data that's read or written isn't visible at the point where a subprogram is declared — examples are dependencies on data, for which we use the `Global` contract, and on flow, for which we use the `Depends` contract.
- reduce the number of variables that need to be considered in flow analysis and proof, a reduction which may be critical in order to scale the analysis to programs with thousands of global variables.

62.1 What's an Abstraction?

Abstraction is an important part of programming language design. It provides two views of the same object: an abstract one and a refined one. The abstract one — usually called *specification* — describes what the object does in a coarse way. A subprogram's specification usually describes how it should be called (e.g., parameter information such as how many and of what types) as well as what it does (e.g., returns a result or modifies one or more of its parameters).

Contract-based programming, as supported in Ada, allows contracts to be added to a subprogram's specification. You use contracts to describe the subprogram's behavior in a more fine-grained manner, but all the details of how the subprogram actually works are left to its refined view, its implementation.

Take a look at the example code shown below.

Listing 1867: `increase.ads`

```
1 procedure Increase (X : in out Integer) with
2   Global => null,
3   Pre    => X <= 100,
4   Post   => X'Old < X;
```

Listing 1868: `increase.adb`

```
1 procedure Increase (X : in out Integer) is
2 begin
3   X := X + 1;
4 end Increase;
```


Code block metadata

```
Project: Courses.Intro_To_Spark.State_Abstraction.No_Abstraction
MD5: c4c8f229aeb1b5c12744d26369a8603f
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
increase.adb:3:11: info: overflow check proved
increase.ads:2:03: info: data dependencies proved
increase.ads:4:13: info: postcondition proved
```

We've written a specification of the subprogram `Increase` to say that it's called with a single argument, a variable of type **Integer** whose initial value is less than 100. Our contract says that the only effect of the subprogram is to increase the value of its argument.

62.2 Why is Abstraction Useful?

A good abstraction of a subprogram's implementation is one whose specification precisely and completely summarizes what its callers can rely on. In other words, a caller of that subprogram shouldn't rely on any behavior of its implementation if that behavior isn't documented in its specification.

For example, callers of the subprogram `Increase` can assume that it always strictly increases the value of its argument. In the code snippet shown below, this means the loop must terminate.

Listing 1869: `increase.ads`

```
1 procedure Increase (X : in out Integer) with
2   Global => null,
3   Pre    => X <= 100,
4   Post   => X'Old < X;
```

Listing 1870: `client.adb`

```
1 with Increase;
2 procedure Client is
3   X : Integer := 0;
4 begin
5   while X <= 100 loop      -- The loop will terminate
6     Increase (X);         -- Increase can be called safely
7   end loop;
8   pragma Assert (X = 101); -- Will this hold?
9 end Client;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.State_Abstraction.Using_Abstraction
MD5: 9cd07cb04ae2194343931f0561693be4
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
```

(continues on next page)

(continued from previous page)

```

Phase 3 of 3: flow analysis and proof ...
client.adb:6:07: warning: no Always_Terminates aspect available for "Increase"
↳[assumed-always-terminates]
client.adb:6:07: warning: assuming "Increase" always terminates
client.adb:8:19: medium: assertion might fail
gnatprove: unproved check messages considered as errors

```

Callers can also assume that the implementation of `Increase` won't cause any runtime errors when called in the loop. On the other hand, nothing in the specification guarantees that the assertion shown above is correct: it may fail if `Increase`'s implementation is changed.

If you follow this basic principle, abstraction can bring you significant benefits. It simplifies both your program's implementation and verification. It also makes maintenance and code reuse much easier since changes to the implementation of an object shouldn't affect the code using this object. Your goal in using it is that it should be enough to understand the specification of an object in order to use that object, since understanding the specification is usually much simpler than understanding the implementation.

GNATprove relies on the abstraction defined by subprogram contracts and therefore doesn't prove the assertion after the loop in `Client` above.

62.3 Abstraction of a Package's State

Subprograms aren't the only objects that benefit from abstraction. The state of a package — the set of persistent variables defined in it — can also be hidden from external users. You achieve this form of abstraction — called *state abstraction* — by defining variables in the body or private part of a package so they can only be accessed through subprogram calls. For example, our `Stack` package shown below provides an abstraction for a `Stack` object which can only be modified using the `Pop` and `Push` procedures.

```

package Stack is
  procedure Pop (E : out Element);
  procedure Push (E : in Element);
end Stack;

package body Stack is
  Content : Element_Array (1 .. Max);
  Top      : Natural;
  ...
end Stack;

```

The fact that we implemented it using an array is irrelevant to the caller. We could change that without impacting our callers' code.

62.4 Declaring a State Abstraction

Hidden state influences a program's behavior, so SPARK allows that state to be declared. You can use the `Abstract_State` aspect, an abstraction that names a state, to do this, but you aren't required to use it even for a package with hidden state. You can use several state abstractions to declare the hidden state of a single package or you can use it for a package with no hidden state at all. However, since SPARK doesn't allow aliasing, different state abstractions must always refer to disjoint sets of variables. A state abstraction isn't a variable: it doesn't have a type and can't be used inside expressions, either those in bodies or contracts.

As an example of the use of this aspect, we can optionally define a state abstraction for the entire hidden state of the `Stack` package like this:

```
package Stack with
  Abstract_State => The_Stack
is
  ...
```

Alternatively, we can define a state abstraction for each hidden variable:

```
package Stack with
  Abstract_State => (Top_State, Content_State)
is
  ...
```

Remember: a state abstraction isn't a variable (it has no type) and can't be used inside expressions. For example:

```
pragma Assert (Stack.Top_State = ...);
-- compilation error: Top_State is not a variable
```

62.5 Refining an Abstract State

Once you've declared an abstract state in a package, you must refine it into its constituents using a Refined_State aspect. You must place the Refined_State aspect on the package body even if the package wouldn't otherwise have required a body. For each state abstraction you've declared for the package, you list the set of variables represented by that state abstraction in its refined state.

If you specify an abstract state for a package, it must be complete, meaning you must have listed every hidden variable as part of some state abstraction. For example, we must add a Refined_State aspect on our Stack package's body linking the state abstraction (The_Stack) to the entire hidden state of the package, which consists of both Content and Top.

Listing 1871: stack.ads

```
1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element);
7   procedure Push (E : Element);
8
9 end Stack;
```

Listing 1872: stack.adb

```
1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;
5
6   type Element_Array is array (1 .. Max) of Element;
7
8   Content : Element_Array := (others => 0);
9   Top      : Natural range 0 .. Max := 0;
10  -- Both Content and Top must be listed in the list of
11  -- constituents of The_Stack
12
13  procedure Pop (E : out Element) is
```

(continues on next page)

(continued from previous page)

```

14  begin
15      E := Content (Top);
16      Top := Top - 1;
17  end Pop;
18
19  procedure Push (E : Element) is
20  begin
21      Top := Top + 1;
22      Content (Top) := E;
23  end Push;
24
25  end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Refined_State
 MD5: 3a794c7a4e4920dab7d01248e50901ab

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 stack.ads:6:20: info: initialization of "E" proved

62.6 Representing Private Variables

You can refine state abstractions in the package body, where all the variables are visible. When only the package's specification is available, you need a way to specify which state abstraction each private variable belongs to. You do this by adding the `Part_Of` aspect to the variable's declaration.

`Part_Of` annotations are mandatory: if you gave a package an abstract state annotation, you must link all the hidden variables defined in its private part to a state abstraction. For example:

Listing 1873: stack.ads

```

1  package Stack with
2      Abstract_State => The_Stack
3  is
4      type Element is new Integer;
5
6      procedure Pop (E : out Element);
7      procedure Push (E : Element);
8
9  private
10
11      Max : constant := 100;
12
13      type Element_Array is array (1 .. Max) of Element;
14
15      Content : Element_Array with Part_Of => The_Stack;
16      Top : Natural range 0 .. Max with Part_Of => The_Stack;
17
18  end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Private_Variables
MD5: 3b5f7edca8a4511071d2397197b01fda

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

Since we chose to define Content and Top in Stack's private part instead of its body, we had to add a Part_Of aspect to both of their declarations, associating them with the state abstraction The_Stack, even though it's the only state abstraction. However, we still need to list them in the Refined_State aspect in Stack's body.

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

62.7 Additional State

62.7.1 Nested Packages

So far, we've only discussed hidden variables. But variables aren't the only component of a package's state. If a package P contains a nested package, the nested package's state is also part of P's state. If the nested package is hidden, its state is part of P's hidden state and must be listed in P's state refinement.

We see this in the example below, where the package Hidden_Nested's hidden state is part of P's hidden state.

Listing 1874: p.ads

```
1 package P with
2   Abstract_State => State
3 is
4   package Visible_Nested with
5     Abstract_State => Visible_State
6   is
7     procedure Get (E : out Integer);
8   end Visible_Nested;
9 end P;
```

Listing 1875: p.adb

```
1 package body P with
2   Refined_State => (State => Hidden_Nested.Hidden_State)
3 is
4   package Hidden_Nested with
5     Abstract_State => Hidden_State,
6     Initializes    => Hidden_State
7   is
8     function Get return Integer;
9   end Hidden_Nested;
10
11   package body Hidden_Nested with
12     Refined_State => (Hidden_State => Cnt)
13   is
14     Cnt : Integer := 0;
15
16     function Get return Integer is (Cnt);
```

(continues on next page)

(continued from previous page)

```

17   end Hidden_Nested;
18
19   package body Visible_Nested with
20     Refined_State => (Visible_State => Checked)
21   is
22     Checked : Boolean := False;
23
24     procedure Get (E : out Integer) is
25     begin
26       Checked := True;
27       E := Hidden_Nested.Get;
28     end Get;
29   end Visible_Nested;
30 end P;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Nested_Packages
 MD5: 8260089cbd651de296dd790506c76fd8

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
p.adb:6:07: info: flow dependencies proved
p.adb:8:16: info: implicit aspect Always_Terminates on "Get" has been proved,
↳subprogram will terminate
p.ads:7:22: info: initialization of "E" proved

```

Any visible state of Hidden_Nested would also have been part of P's hidden state. However, if P contains a visible nested package, that nested package's state isn't part of P's hidden state. Instead, you should declare that package's hidden state in a separate state abstraction on its own declaration, like we did above for Visible_Nested.

62.7.2 Constants that Depend on Variables

Some constants are also possible components of a state abstraction. These are constants whose value depends either on a variable or a subprogram parameter. They're handled as variables during flow analysis because they participate in the flow of information between variables throughout the program. Therefore, GNATprove considers these constants to be part of a package's state just like it does for variables.

If you've specified a state abstraction for a package, you must list such hidden constants declared in that package in the state abstraction refinement. However, constants that don't depend on variables don't participate in the flow of information and must not appear in a state refinement.

Let's look at this example.

Listing 1876: stack.ads

```

1   package Stack with
2     Abstract_State => The_Stack
3   is
4     type Element is new Integer;
5
6     procedure Pop (E : out Element);
7     procedure Push (E : Element);
8   end Stack;

```

Listing 1877: configuration.ads

```

1 package Configuration with
2   Initializes => External_Variable
3 is
4   External_Variable : Positive with Volatile;
5 end Configuration;
```

Listing 1878: stack.adb

```

1 with Configuration;
2 pragma Elaborate (Configuration);
3
4 package body Stack with
5   Refined_State => (The_Stack => (Content, Top, Max))
6   -- Max has variable inputs. It must appear as a
7   -- constituent of The_Stack
8 is
9   Max : constant Positive := Configuration.External_Variable;
10
11   type Element_Array is array (1 .. Max) of Element;
12
13   Content : Element_Array := (others => 0);
14   Top      : Natural range 0 .. Max := 0;
15
16   procedure Pop (E : out Element) is
17   begin
18     E := Content (Top);
19     Top := Top - 1;
20   end Pop;
21
22   procedure Push (E : Element) is
23   begin
24     Top := Top + 1;
25     Content (Top) := E;
26   end Push;
27
28 end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Constants_And_Variables
 MD5: 109a6340ef0f3b0dc88e0fe5888b9a53

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 configuration.ads:2:03: info: flow dependencies proved
 stack.ads:6:20: info: initialization of "E" proved

Here, Max — the maximum number of elements that can be stored in the stack — is initialized from a variable in an external package. Because of this, we must include Max as part of the state abstraction The_Stack.

Note

For more details on state abstractions, see the [SPARK User's Guide](#)⁴⁰⁵.

⁴⁰⁵ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/package_contracts.html#

62.8 Subprogram Contracts

62.8.1 Global and Depends

Hidden variables can only be accessed through subprogram calls, so you document how state abstractions are modified during the program's execution via the contracts of those subprograms. You use `Global` and `Depends` contracts to specify which of the state abstractions are used by a subprogram and how values flow through the different variables. The `Global` and `Depends` contracts that you write when referring to state abstractions are often less precise than contracts referring to visible variables since the possibly different dependencies of the hidden variables contained within a state abstraction are collapsed into a single dependency.

Let's add `Global` and `Depends` contracts to the `Pop` procedure in our stack.

Listing 1879: stack.ads

```

1 package Stack with
2   Abstract_State => (Top_State, Content_State)
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element) with
7     Global  => (Input  => Content_State,
8               In_Out => Top_State),
9     Depends => (Top_State => Top_State,
10              E          => (Content_State, Top_State));
11
12 end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Global_Depends
MD5: a7b383c35508d6a8294bf7cf0fe332ac

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of `Global` contracts ...
Phase 3 of 3: analysis of data and information flow ...

In this example, the `Pop` procedure only modifies the value of the hidden variable `Top`, while `Content` is unchanged. By using distinct state abstractions for the two variables, we're able to preserve this semantic in the contract.

Let's contrast this example with a different representation of `Global` and `Depends` contracts, this time using a single abstract state.

Listing 1880: stack.ads

```

1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element) with
7     Global  => (In_Out => The_Stack),
8     Depends => ((The_Stack, E) => The_Stack);
9
10 end Stack;
```

state-abstraction

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Global_Single_Abstract_State
MD5: f89f6026fa5ee3c18baf0af9d7c3dbca

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

Here, `Top_State` and `Content_State` are merged into a single state abstraction, `The_Stack`. By doing so, we've hidden the fact that `Content` isn't modified (though we're still showing that `Top` may be modified). This loss in precision is reasonable here, since it's the whole point of the abstraction. However, you must be careful not to aggregate unrelated hidden state because this risks their annotations becoming meaningless.

Even though imprecise contracts that consider state abstractions as a whole are perfectly reasonable for users of a package, you should write `Global` and `Depends` contracts that are as precise as possible within the package body. To allow this, SPARK introduces the notion of *refined contracts*, which are precise contracts specified on the bodies of subprograms where state refinements are visible. These contracts are the same as normal `Global` and `Depends` contracts except they refer directly to the hidden state of the package.

When a subprogram is called inside the package body, you should write refined contracts instead of the general ones so that the verification can be as precise as possible. However, refined `Global` and `Depends` are optional: if you don't specify them, GNATprove will compute them to check the package's implementation.

For our `Stack` example, we could add refined contracts as shown below.

Listing 1881: `stack.ads`

```
1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element) with
7     Global => (In_Out => The_Stack),
8     Depends => ((The_Stack, E) => The_Stack);
9
10  procedure Push (E : Element) with
11    Global => (In_Out => The_Stack),
12    Depends => (The_Stack => (The_Stack, E));
13
14 end Stack;
```

Listing 1882: `stack.adb`

```
1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;
5
6   type Element_Array is array (1 .. Max) of Element;
7
8   Content : Element_Array := (others => 0);
9   Top : Natural range 0 .. Max := 0;
10
11  procedure Pop (E : out Element) with
12    Refined_Global => (Input => Content,
```

(continues on next page)

(continued from previous page)

```

13         In_Out => Top),
14     Refined_Depends => (Top => Top,
15                         E   => (Content, Top))
16 is
17 begin
18     E := Content (Top);
19     Top := Top - 1;
20 end Pop;
21
22 procedure Push (E : Element) with
23     Refined_Global => (In_Out => (Content, Top)),
24     Refined_Depends => (Content =>+ (Content, Top, E),
25                       Top      => Top) is
26 begin
27     Top := Top + 1;
28     Content (Top) := E;
29 end Push;
30
31 end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Global_Refined
MD5: b7e700645885155ea7faf2f4170f0462

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

62.8.2 Preconditions and Postconditions

We mostly express functional properties of subprograms using preconditions and postconditions. These are standard Boolean expressions, so they can't directly refer to state abstractions. To work around this restriction, we can define functions to query the value of hidden variables. We then use these functions in place of the state abstraction in the contract of other subprograms.

For example, we can query the state of the stack with functions `Is_Empty` and `Is_Full` and call these in the contracts of procedures `Pop` and `Push`:

Listing 1883: stack.ads

```

1 package Stack is
2     type Element is new Integer;
3
4     function Is_Empty return Boolean;
5     function Is_Full  return Boolean;
6
7     procedure Pop (E : out Element) with
8         Pre => not Is_Empty,
9         Post => not Is_Full;
10
11     procedure Push (E : Element) with
12         Pre => not Is_Full,
13         Post => not Is_Empty;
14
15 end Stack;

```

Listing 1884: stack.adb

```

1 package body Stack is
2
3   Max : constant := 100;
4
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array := (others => 0);
8   Top      : Natural range 0 .. Max := 0;
9
10  function Is_Empty return Boolean is (Top = 0);
11  function Is_Full  return Boolean is (Top = Max);
12
13  procedure Pop (E : out Element) is
14  begin
15      E := Content (Top);
16      Top := Top - 1;
17  end Pop;
18
19  procedure Push (E : Element) is
20  begin
21      Top := Top + 1;
22      Content (Top) := E;
23  end Push;
24
25 end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Pre_Postconditions_1
MD5: fe9d4b65ba1beeabc7cf0feda29b8b3c

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
stack.adb:15:23: info: index check proved
stack.adb:16:18: info: range check proved
stack.adb:21:28: info: range check proved
stack.adb:22:16: info: index check proved
stack.ads:4:13: info: implicit aspect Always_Terminates on "Is_Empty" has been
↳proved, subprogram will terminate
stack.ads:5:13: info: implicit aspect Always_Terminates on "Is_Full" has been
↳proved, subprogram will terminate
stack.ads:7:19: info: initialization of "E" proved
stack.ads:9:14: info: postcondition proved
stack.ads:13:14: info: postcondition proved

```

Just like we saw for Global and Depends contracts, you may often find it useful to have a more precise view of functional contracts in the context where the hidden variables are visible. You do this using expression functions in the same way we did for the functions Is_Empty and Is_Full above. As expression function, bodies act as contracts for GNATprove, so they automatically give a more precise version of the contracts when their implementation is visible.

You may often need a more constraining contract to verify the package's implementation but want to be less strict outside the abstraction. You do this using the Refined_Post aspect. This aspect, when placed on a subprogram's body, provides stronger guarantees to internal callers of a subprogram. If you provide one, the refined postcondition must imply the subprogram's postcondition. This is checked by GNATprove, which reports a failing

postcondition if the refined postcondition is too weak, even if it's actually implied by the subprogram's body. SPARK doesn't perform a similar verification for normal preconditions.

For example, we can refine the postconditions in the bodies of Pop and Push to be more detailed than what we wrote for them in their specification.

Listing 1885: stack.ads

```

1 package Stack is
2   type Element is new Integer;
3
4   function Is_Empty return Boolean;
5   function Is_Full  return Boolean;
6
7   procedure Pop (E : out Element) with
8     Pre  => not Is_Empty,
9     Post => not Is_Full;
10
11  procedure Push (E : Element) with
12    Pre  => not Is_Full,
13    Post => not Is_Empty;
14
15 end Stack;
```

Listing 1886: stack.adb

```

1 package body Stack is
2
3   Max : constant := 100;
4
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array := (others => 0);
8   Top      : Natural range 0 .. Max := 0;
9
10  function Is_Empty return Boolean is (Top = 0);
11  function Is_Full  return Boolean is (Top = Max);
12
13  procedure Pop (E : out Element) with
14    Refined_Post => not Is_Full and E = Content (Top)'Old
15  is
16  begin
17    E := Content (Top);
18    Top := Top - 1;
19  end Pop;
20
21  procedure Push (E : Element) with
22    Refined_Post => not Is_Empty and E = Content (Top)
23  is
24  begin
25    Top := Top + 1;
26    Content (Top) := E;
27  end Push;
28
29 end Stack;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Pre_Postconditions_2
 MD5: 4691565d58ba039b3cbd06e65cecfa88

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
stack.adb:14:22: info: refined post proved
stack.adb:14:51: info: index check proved
stack.adb:17:23: info: index check proved
stack.adb:18:18: info: range check proved
stack.adb:22:22: info: refined post proved
stack.adb:22:52: info: index check proved
stack.adb:25:28: info: range check proved
stack.adb:26:16: info: index check proved
stack.ads:4:13: info: implicit aspect Always_Terminates on "Is_Empty" has been_
↳proved, subprogram will terminate
stack.ads:5:13: info: implicit aspect Always_Terminates on "Is_Full" has been_
↳proved, subprogram will terminate
stack.ads:7:19: info: initialization of "E" proved
stack.ads:9:14: info: postcondition proved
stack.ads:13:14: info: postcondition proved

```

Note

For more details on refinement in contracts, see the [SPARK User's Guide](#)⁴⁰⁶.

62.9 Initialization of Local Variables

As part of flow analysis, GNATprove checks for the proper initialization of variables. Therefore, flow analysis needs to know which variables are initialized during the package's elaboration.

You can use the `Initializes` aspect to specify the set of visible variables and state abstractions that are initialized during the elaboration of a package. An `Initializes` aspect can't refer to a variable that isn't defined in the unit since, in SPARK, a package can only initialize variables declared immediately within the package.

`Initializes` aspects are optional. If you don't supply any, they'll be derived by GNATprove. For our Stack example, we could add an `Initializes` aspect.

Listing 1887: stack.ads

```

1 package Stack with
2   Abstract_State => The_Stack,
3   Initializes   => The_Stack
4 is
5   type Element is new Integer;
6
7   procedure Pop (E : out Element);
8
9 end Stack;

```

Listing 1888: stack.adb

```

1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;

```

(continues on next page)

⁴⁰⁶ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#state-abstraction-and-contracts

(continued from previous page)

```

5
6  type Element_Array is array (1 .. Max) of Element;
7
8  Content : Element_Array := (others => 0);
9  Top      : Natural range 0 .. Max := 0;
10
11  procedure Pop (E : out Element) is
12  begin
13      E := Content (Top);
14      Top := Top - 1;
15  end Pop;
16
17 end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Local_Init
MD5: 710e74959fd2ef8f5089c4636d7ec13b

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
stack.ads:3:03: info: flow dependencies proved
stack.ads:7:20: info: initialization of "E" proved

Flow analysis also checks for dependencies between variables, so it must be aware of how information flows through the code that performs the initialization of states. We discussed one use of the `Initializes` aspect above. But you also can use it to provide flow information. If the initial value of a variable or state abstraction is dependent on the value of another visible variable or state abstraction from another package, you must list this dependency in the `Initializes` contract. You specify the list of entities on which a variable's initial value depends using an arrow following that variable's name.

Let's look at this example:

Listing 1889: q.ads

```

1  package Q is
2      External_Variable : Integer := 2;
3  end Q;

```

Listing 1890: p.ads

```

1  with Q;
2  package P with
3      Initializes => (V1, V2 => Q.External_Variable)
4  is
5      V1 : Integer := 0;
6      V2 : Integer := Q.External_Variable;
7  end P;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Initializes
MD5: c8aa7f21729f3b926bf3d25a826cccb2

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
p.ads:3:03: info: flow dependencies proved
```

Here we indicated that V2's initial value depends on the value of Q.External_Variable by including that dependency in the Initializes aspect of P. We didn't list any dependency for V1 because its initial value doesn't depend on any external variable. We could also have stated that lack of dependency explicitly by writing V1 => **null**.

GNATprove computes dependencies of initial values if you don't supply an Initializes aspect. However, if you do provide an Initializes aspect for a package, it must be complete: you must list every initialized state of the package, along with all its external dependencies.

Note

For more details on Initializes, see the [SPARK User's Guide](#)⁴⁰⁷.

62.10 Code Examples / Pitfalls

This section contains some code examples to illustrate potential pitfalls.

62.10.1 Example #1

Package Communication defines a hidden local package, Ring_Buffer, whose capacity is initialized from an external configuration during elaboration.

Listing 1891: configuration.ads

```
1 package Configuration is
2
3     External_Variable : Natural := 1;
4
5 end Configuration;
```

Listing 1892: communication.ads

```
1 with Configuration;
2
3 package Communication with
4     Abstract_State => State,
5     Initializes    => (State => Configuration.External_Variable)
6 is
7     function Get_Capacity return Natural;
8
9 private
10
11     package Ring_Buffer with
12         Initializes => (Capacity => Configuration.External_Variable)
13     is
14         Capacity : constant Natural := Configuration.External_Variable;
15         end Ring_Buffer;
16
17 end Communication;
```

⁴⁰⁷ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/package_contracts.html#package-initialization

Listing 1893: communication.adb

```

1 package body Communication with
2   Refined_State => (State => Ring_Buffer.Capacity)
3 is
4
5   function Get_Capacity return Natural is
6   begin
7     return Ring_Buffer.Capacity;
8   end Get_Capacity;
9
10 end Communication;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_01
MD5: 207e999f85a5b39fa2b9aebbc836b479

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
communication.adb:2:41: error: "Capacity" cannot act as constituent of state "State"
↪ "
communication.adb:2:41: error: missing Part_Of indicator at communication.ads:14_
↪ should specify encapsulator "State"
gnatprove: error during generation of Global contracts
```

This example isn't correct. Capacity is declared in the private part of Communication. Therefore, we should have linked it to State by using the Part_Of aspect in its declaration.

62.10.2 Example #2

Let's add Part_Of to the state of hidden local package Ring_Buffer, but this time we hide variable Capacity inside the private part of Ring_Buffer.

Listing 1894: configuration.ads

```

1 package Configuration is
2
3   External_Variable : Natural := 1;
4
5 end Configuration;
```

Listing 1895: communication.ads

```

1 with Configuration;
2
3 package Communication with
4   Abstract_State => State
5 is
6   private
7
8     package Ring_Buffer with
9       Abstract_State => (B_State with Part_Of => State),
10      Initializes    => (B_State => Configuration.External_Variable)
11     is
12       function Get_Capacity return Natural;
13     private
14       Capacity : constant Natural := Configuration.External_Variable
```

(continues on next page)

(continued from previous page)

```

15     with Part_Of => B_State;
16 end Ring_Buffer;
17
18 end Communication;
```

Listing 1896: communication.adb

```

1 package body Communication with
2   Refined_State => (State => Ring_Buffer.B_State)
3 is
4
5   package body Ring_Buffer with
6     Refined_State => (B_State => Capacity)
7   is
8     function Get_Capacity return Natural is (Capacity);
9   end Ring_Buffer;
10
11 end Communication;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_02
 MD5: b8d31fcfb11bf305646efe07baeb91b

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
communication.ads:10:06: info: flow dependencies proved
communication.ads:12:16: info: implicit aspect Always_Terminates on "Get_Capacity"
↳ has been proved, subprogram will terminate
```

This program is correct and GNATprove is able to verify it.

62.10.3 Example #3

Package Counting defines two counters: Black_Counter and Red_Counter. It provides separate initialization procedures for each, both called from the main procedure.

Listing 1897: counting.ads

```

1 package Counting with
2   Abstract_State => State
3 is
4   procedure Reset_Black_Count;
5   procedure Reset_Red_Count;
6 end Counting;
```

Listing 1898: counting.adb

```

1 package body Counting with
2   Refined_State => (State => (Black_Counter, Red_Counter))
3 is
4   Black_Counter, Red_Counter : Natural;
5
6   procedure Reset_Black_Count is
7   begin
8     Black_Counter := 0;
```

(continues on next page)

(continued from previous page)

```

9   end Reset_Black_Count;
10
11  procedure Reset_Red_Count is
12  begin
13      Red_Counter := 0;
14  end Reset_Red_Count;
15 end Counting;

```

Listing 1899: main.adb

```

1 with Counting; use Counting;
2
3 procedure Main is
4 begin
5     Reset_Black_Count;
6     Reset_Red_Count;
7 end Main;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_03
MD5: bc2d7ccd7419d34f7156a16dfc484229

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
main.adb:5:04: medium: "Counting.State" might not be initialized after elaboration,
↳ of main program "Main"
counting.ads:2:21: warning: no subprogram exists that can initialize abstract
↳ state "Counting.State"
gnatprove: unproved check messages considered as errors

```

This program doesn't read any uninitialized data, but GNATprove fails to verify that. This is because we provided a state abstraction for package Counting, so flow analysis computes the effects of subprograms in terms of this state abstraction and thus considers State to be an in-out global consisting of both Black_Counter and Red_Counter. So it issues the message requiring that State be initialized after elaboration as well as the warning that no procedure in package Counting can initialize its state.

62.10.4 Example #4

Let's remove the abstract state on package Counting.

Listing 1900: counting.ads

```

1 package Counting is
2     procedure Reset_Black_Count;
3     procedure Reset_Red_Count;
4 end Counting;

```

Listing 1901: counting.adb

```

1 package body Counting is
2     Black_Counter, Red_Counter : Natural;
3
4     procedure Reset_Black_Count is
5     begin

```

(continues on next page)

(continued from previous page)

```

6      Black_Counter := 0;
7  end Reset_Black_Count;
8
9  procedure Reset_Red_Count is
10 begin
11     Red_Counter := 0;
12 end Reset_Red_Count;
13 end Counting;

```

Listing 1902: main.adb

```

1 with Counting; use Counting;
2
3 procedure Main is
4 begin
5     Reset_Black_Count;
6     Reset_Red_Count;
7 end Main;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_04
MD5: 3ddd934b6ede6df7b823e46828694d12

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

This example is correct. Because we didn't provide a state abstraction, GNATprove reasons in terms of variables, instead of states, and proves data initialization without any problem.

62.10.5 Example #5

Let's restore the abstract state to package Counting, but this time provide a procedure Reset_All that calls the initialization procedures Reset_Black_Counter and Reset_Red_Counter.

Listing 1903: counting.ads

```

1 package Counting with
2   Abstract_State => State
3 is
4   procedure Reset_Black_Count with Global => (In_Out => State);
5   procedure Reset_Red_Count   with Global => (In_Out => State);
6   procedure Reset_All         with Global => (Output => State);
7 end Counting;

```

Listing 1904: counting.adb

```

1 package body Counting with
2   Refined_State => (State => (Black_Counter, Red_Counter))
3 is
4   Black_Counter, Red_Counter : Natural;
5
6   procedure Reset_Black_Count is
7   begin
8     Black_Counter := 0;

```

(continues on next page)

(continued from previous page)

```

9   end Reset_Black_Count;
10
11  procedure Reset_Red_Count is
12  begin
13      Red_Counter := 0;
14  end Reset_Red_Count;
15
16  procedure Reset_All is
17  begin
18      Reset_Black_Count;
19      Reset_Red_Count;
20  end Reset_All;
21 end Counting;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_05
MD5: d123ccc644fe6999699388708f2ecf89

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
counting.ads:4:37: info: data dependencies proved
counting.ads:5:37: info: data dependencies proved
counting.ads:6:14: info: initialization of "Black_Counter" constituent of "State"
↳proved
counting.ads:6:14: info: initialization of "Red_Counter" constituent of "State"
↳proved
counting.ads:6:37: info: data dependencies proved

```

This example is correct. Flow analysis computes refined versions of Global contracts for internal calls and uses these to verify that Reset_All indeed properly initializes State. The Refined_Global and Global annotations are not mandatory and can be computed by GNATprove.

62.10.6 Example #6

Let's consider yet another version of our abstract stack unit.

Listing 1905: stack.ads

```

1  package Stack with
2      Abstract_State => The_Stack
3  is
4      pragma Unevaluated_Use_Of_Old (Allow);
5
6      type Element is new Integer;
7
8      type Element_Array is array (Positive range <>) of Element;
9      Max : constant Natural := 100;
10     subtype Length_Type is Natural range 0 .. Max;
11
12     procedure Push (E : Element) with
13         Post =>
14             not Is_Empty and
15             (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);
16
17     function Peek      return Element with Pre => not Is_Empty;

```

(continues on next page)

(continued from previous page)

```

18     function Is_Full   return Boolean;
19     function Is_Empty return Boolean;
20 end Stack;

```

Listing 1906: stack.adb

```

1  package body Stack with
2    Refined_State => (The_Stack => (Top, Content))
3  is
4    Top      : Length_Type := 0;
5    Content : Element_Array (1 .. Max) := (others => 0);
6
7    procedure Push (E : Element) is
8    begin
9      Top      := Top + 1;
10     Content (Top) := E;
11 end Push;
12
13 function Peek      return Element is (Content (Top));
14 function Is_Full   return Boolean is (Top >= Max);
15 function Is_Empty return Boolean is (Top = 0);
16 end Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_06
MD5: 9da2b74da203a639dc66b2d33cbd500d

Build output

```

stack.ads:15:39: error: there is no applicable operator "=" for package or
↳ procedure name
gprbuild: *** compilation phase failed

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
stack.ads:15:39: error: there is no applicable operator "=" for package or
↳ procedure name
gnatprove: error during generation of Global contracts

```

This example isn't correct. There's a compilation error in Push's postcondition: The_Stack is a state abstraction, not a variable, and therefore can't be used in an expression.

62.10.7 Example #7

In this version of our abstract stack unit, a copy of the stack is returned by function Get_Stack, which we call in the postcondition of Push to specify that the stack shouldn't be modified if it's full. We also assert that after we push an element on the stack, either the stack is unchanged (if it was already full) or its top element is equal to the element just pushed.

Listing 1907: stack.ads

```

1  package Stack with
2    Abstract_State => The_Stack
3  is
4    pragma Unevaluated_Use_Of_Old (Allow);
5

```

(continues on next page)

(continued from previous page)

```

6  type Stack_Model is private;
7
8  type Element is new Integer;
9  type Element_Array is array (Positive range <>) of Element;
10 Max : constant Natural := 100;
11 subtype Length_Type is Natural range 0 .. Max;
12
13 function Peek      return Element with Pre => not Is_Empty;
14 function Is_Full   return Boolean;
15 function Is_Empty  return Boolean;
16 function Get_Stack return Stack_Model;
17
18 procedure Push (E : Element) with
19   Post => not Is_Empty and
20     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
21
22 private
23
24   type Stack_Model is record
25     Top      : Length_Type := 0;
26     Content : Element_Array (1 .. Max) := (others => 0);
27   end record;
28
29 end Stack;

```

Listing 1908: stack.adb

```

1  package body Stack with
2    Refined_State => (The_Stack => (Top, Content))
3  is
4    Top      : Length_Type := 0;
5    Content : Element_Array (1 .. Max) := (others => 0);
6
7    procedure Push (E : Element) is
8    begin
9      if Top >= Max then
10         return;
11       end if;
12       Top      := Top + 1;
13       Content (Top) := E;
14     end Push;
15
16     function Peek      return Element is (Content (Top));
17     function Is_Full   return Boolean is (Top >= Max);
18     function Is_Empty  return Boolean is (Top = 0);
19
20     function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
21
22 end Stack;

```

Listing 1909: use_stack.adb

```

1  with Stack; use Stack;
2
3  procedure Use_Stack (E : Element) with
4    Pre => not Is_Empty
5  is
6    F : Element := Peek;
7  begin
8    Push (E);

```

(continues on next page)

(continued from previous page)

```

9   pragma Assert (Peek = E or Peek = F);
10  end Use_Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_07
MD5: 4831aa7f018f2e2d4e6d102095f8f631

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
use_stack.adb:9:19: medium: assertion might fail [possible fix: precondition of
↳subprogram at line 3 should mention E]
gnatprove: unproved check messages considered as errors

```

This program is correct, but GNATprove can't prove the assertion in Use_Stack. Indeed, even if Get_Stack is an expression function, its body isn't visible outside of Stack's body, where it's defined.

62.10.8 Example #8

Let's move the definition of Get_Stack and other expression functions inside the private part of the spec of Stack.

Listing 1910: stack.ads

```

1  package Stack with
2    Abstract_State => The_Stack
3  is
4    pragma Unevaluated_Use_Of_Old (Allow);
5
6    type Stack_Model is private;
7
8    type Element is new Integer;
9    type Element_Array is array (Positive range <>) of Element;
10   Max : constant Natural := 100;
11   subtype Length_Type is Natural range 0 .. Max;
12
13   function Peek      return Element with Pre => not Is_Empty;
14   function Is_Full   return Boolean;
15   function Is_Empty  return Boolean;
16   function Get_Stack return Stack_Model;
17
18   procedure Push (E : Element) with
19     Post => not Is_Empty and
20       (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
21
22 private
23
24   Top      : Length_Type := 0 with Part_Of => The_Stack;
25   Content : Element_Array (1 .. Max) := (others => 0) with
26     Part_Of => The_Stack;
27
28   type Stack_Model is record
29     Top      : Length_Type := 0;
30     Content : Element_Array (1 .. Max) := (others => 0);
31   end record;
32

```

(continues on next page)

(continued from previous page)

```

33  function Peek      return Element    is (Content (Top));
34  function Is_Full   return Boolean    is (Top >= Max);
35  function Is_Empty  return Boolean    is (Top = 0);
36
37  function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
38
39  end Stack;

```

Listing 1911: stack.adb

```

1  package body Stack with
2    Refined_State => (The_Stack => (Top, Content))
3  is
4
5    procedure Push (E : Element) is
6    begin
7      if Top >= Max then
8        return;
9      end if;
10     Top           := Top + 1;
11     Content (Top) := E;
12   end Push;
13
14   end Stack;

```

Listing 1912: use_stack.adb

```

1  with Stack; use Stack;
2
3  procedure Use_Stack (E : Element) with
4    Pre => not Is_Empty
5  is
6    F : Element := Peek;
7  begin
8    Push (E);
9    pragma Assert (Peek = E or Peek = F);
10   end Use_Stack;

```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_08
MD5: 7e5204d3f69e71c212e7263906a89da4

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
use_stack.adb:6:19: info: precondition proved
use_stack.adb:9:19: info: precondition proved
use_stack.adb:9:19: info: assertion proved
use_stack.adb:9:31: info: precondition proved
stack.adb:10:30: info: range check proved
stack.adb:11:16: info: index check proved
stack.ads:13:13: info: implicit aspect Always_Terminates on "Peek" has been proved,
↳ subprogram will terminate
stack.ads:14:13: info: implicit aspect Always_Terminates on "Is_Full" has been
↳ proved, subprogram will terminate
stack.ads:15:13: info: implicit aspect Always_Terminates on "Is_Empty" has been
↳ proved, subprogram will terminate

```

(continues on next page)

(continued from previous page)

```
stack.ads:16:13: info: implicit aspect Always_Terminates on "Get_Stack" has been_
↳proved, subprogram will terminate
stack.ads:19:14: info: postcondition proved
stack.ads:20:60: info: precondition proved
stack.ads:33:55: info: index check proved
```

This example is correct. GNATprove can verify the assertion in Use_Stack because it has visibility to Get_Stack's body.

62.10.9 Example #9

Package Data defines three variables, Data_1, Data_2 and Data_3, that are initialized at elaboration (in Data's package body) from an external interface that reads the file system.

Listing 1913: external_interface.ads

```
1 package External_Interface with
2   Abstract_State => File_System,
3   Initializes   => File_System
4 is
5   type Data_Type_1 is new Integer;
6   type Data_Type_2 is new Integer;
7   type Data_Type_3 is new Integer;
8
9   type Data_Record is record
10     Field_1 : Data_Type_1;
11     Field_2 : Data_Type_2;
12     Field_3 : Data_Type_3;
13   end record;
14
15   procedure Read_Data (File_Name : String; Data : out Data_Record)
16     with Global => File_System;
17 end External_Interface;
```

Listing 1914: data.ads

```
1 with External_Interface; use External_Interface;
2
3 package Data with
4   Initializes => (Data_1, Data_2, Data_3)
5 is
6   pragma Elaborate_Body;
7
8   Data_1 : Data_Type_1;
9   Data_2 : Data_Type_2;
10  Data_3 : Data_Type_3;
11
12 end Data;
```

Listing 1915: data.adb

```
1 with External_Interface;
2 pragma Elaborate_All (External_Interface);
3
4 package body Data is
5 begin
6   declare
7     Data_Read : Data_Record;
8   begin
```

(continues on next page)

(continued from previous page)

```

9      Read_Data ("data_file_name", Data_Read);
10     Data_1 := Data_Read.Field_1;
11     Data_2 := Data_Read.Field_2;
12     Data_3 := Data_Read.Field_3;
13 end;
14 end Data;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_09
MD5: 0ca44501f0c991865ea50d2ef663d992

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
data.adb:9:07: warning: no Always_Terminates aspect available for "Read_Data"
  ↳[assumed-always-terminates]
data.adb:9:07: warning: assuming "Read_Data" always terminates
data.adb:9:07: high: "External_Interface.File_System" must be mentioned as an
  ↳input of the Initializes aspect of "Data" (SPARK RM 7.1.5(11))
gnatprove: unproved check messages considered as errors
```

This example isn't correct. The dependency between Data_1's, Data_2's, and Data_3's initial values and File_System must be listed in Data's Initializes aspect.

62.10.10 Example #10

Let's remove the Initializes contract on package Data.

Listing 1916: external_interface.ads

```

1 package External_Interface with
2   Abstract_State => File_System,
3   Initializes   => File_System
4 is
5   type Data_Type_1 is new Integer;
6   type Data_Type_2 is new Integer;
7   type Data_Type_3 is new Integer;
8
9   type Data_Record is record
10     Field_1 : Data_Type_1;
11     Field_2 : Data_Type_2;
12     Field_3 : Data_Type_3;
13   end record;
14
15   procedure Read_Data (File_Name : String; Data : out Data_Record)
16     with Global => File_System;
17 end External_Interface;
```

Listing 1917: data.ads

```

1 with External_Interface; use External_Interface;
2
3 package Data is
4   pragma Elaborate_Body;
5
6   Data_1 : Data_Type_1;
7   Data_2 : Data_Type_2;
```

(continues on next page)

(continued from previous page)

```
8   Data_3 : Data_Type_3;
9
10  end Data;
```

Listing 1918: data.adb

```
1  with External_Interface;
2  pragma Elaborate_All (External_Interface);
3
4  package body Data is
5  begin
6      declare
7          Data_Read : Data_Record;
8      begin
9          Read_Data ("data_file_name", Data_Read);
10         Data_1 := Data_Read.Field_1;
11         Data_2 := Data_Read.Field_2;
12         Data_3 := Data_Read.Field_3;
13     end;
14 end Data;
```

Code block metadata

Project: Courses.Intro_To_Spark.State_Abstraction.Example_10
MD5: 60cba2c920c7b1031d13c82a982ed0e9

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
external_interface.ads:3:03: info: flow dependencies proved
data.adb:7:07: info: initialization of "Data_Read" proved
data.adb:9:07: warning: no Always_Terminates aspect available for "Read_Data"
↳[assumed-always-terminates]
data.adb:9:07: warning: assuming "Read_Data" always terminates
```

This example is correct. Since Data has no Initializes aspect, GNATprove computes the set of variables initialized during its elaboration as well as their dependencies.

PROOF OF FUNCTIONAL CORRECTNESS

This section is dedicated to the functional correctness of programs. It presents advanced proof features that you may need to use for the specification and verification of your program's complex properties.

63.1 Beyond Program Integrity

When we speak about the *correctness* of a program or subprogram, we mean the extent to which it complies with its specification. Functional correctness is specifically concerned with properties that involve the relations between the subprogram's inputs and outputs, as opposed to other properties such as running time or memory consumption.

For functional correctness, we usually specify stronger properties than those required to just prove program integrity. When we're involved in a certification processes, we should derive these properties from the requirements of the system, but, especially in non-certification contexts, they can also come from more informal sources, such as the program's documentation, comments in its code, or test oracles.

For example, if one of our goals is to ensure that no runtime error is raised when using the result of the function `Find` below, it may be enough to know that the result is either 0 or in the range of `A`. We can express this as a postcondition of `Find`.

Listing 1919: `show_find.ads`

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural with
6         Post => Find'Result in 0 | A'Range;
7
8 end Show_Find;
```

Listing 1920: `show_find.adb`

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             if A (I) = E then
7                 return I;
8             end if;
9         end loop;
10        return 0;
11    end Find;
```

(continues on next page)

(continued from previous page)

```

12
13 end Show_Find;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Find_1
MD5: d8f4ace6620fd46af170977c29947289

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_find.adb:7:20: info: range check proved
show_find.ads:5:13: info: implicit aspect Always_Terminates on "Find" has been
→proved, subprogram will terminate
show_find.ads:6:14: info: postcondition proved

```

In this case, it's automatically proved by GNATprove.

However, to be sure that Find performs the task we expect, we may want to verify more complex properties of that function. For example, we want to ensure it returns an index of A where E is stored and returns 0 only if E is nowhere in A. Again, we can express this as a postcondition of Find.

Listing 1921: show_find.ads

```

1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural with
6         Post =>
7         (if (for all I in A'Range => A (I) /= E)
8             then Find'Result = 0
9             else Find'Result in A'Range and then A (Find'Result) = E);
10
11 end Show_Find;

```

Listing 1922: show_find.adb

```

1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             if A (I) = E then
7                 return I;
8             end if;
9         end loop;
10        return 0;
11    end Find;
12
13 end Show_Find;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Find_2
MD5: 8c12b9768228a3ea45ca02199f65057b

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_find.ads:9:14: medium: postcondition might fail, cannot prove Find'Result in A
↳ 'Range
gnatprove: unproved check messages considered as errors

```

This time, GNATprove can't prove this postcondition automatically, but we'll see later that we can help GNATprove by providing a loop invariant, which is checked by GNATprove and allows it to automatically prove the postcondition for Find.

Writing at least part of your program's specification in the form of contracts has many advantages. You can execute those contracts during testing, which improves the maintainability of the code by detecting discrepancies between the program and its specification in earlier stages of development. If the contracts are precise enough, you can use them as oracles to decide whether a given test passed or failed. In that case, they can allow you to verify the outputs of specific subprograms while running a larger block of code. This may, in certain contexts, replace the need for you to perform unit testing, instead allowing you to run integration tests with assertions enabled. Finally, if the code is in SPARK, you can also use GNATprove to formally prove these contracts.

The advantage of a formal proof is that it verifies all possible execution paths, something which isn't always possible by running test cases. For example, during testing, the postcondition of the subprogram Find shown below is checked dynamically for the set of inputs for which Find is called in that test, but just for that set.

Listing 1923: show_find.ads

```

1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural with
6         Post =>
7         (if (for all I in A'Range => A (I) /= E)
8             then Find'Result = 0
9             else Find'Result in A'Range and then A (Find'Result) = E);
10
11 end Show_Find;

```

Listing 1924: show_find.adb

```

1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             if A (I) = E then
7                 return I;
8             end if;
9         end loop;
10        return 0;
11    end Find;
12
13 end Show_Find;

```

Listing 1925: use_find.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Show_Find; use Show_Find;
3

```

(continues on next page)

(continued from previous page)

```

4 procedure Use_Find with
5   SPARK_Mode => Off
6 is
7   Seq : constant Nat_Array (1 .. 3) := (1, 5, 3);
8   Res : Natural;
9 begin
10  Res := Find (Seq, 3);
11  Put_Line ("Found 3 in index #" & Natural'Image (Res) & " of array");
12 end Use_Find;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Find_3
MD5: 05283ef7808ee5d8254cfa4b883e639d

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_find.ads:9:14: medium: postcondition might fail, cannot prove Find'Result in A
↳ 'Range
gnatprove: unproved check messages considered as errors

Runtime output

Found 3 in index # 3 of array

However, if Find is formally verified, that verification checks its postcondition for all possible inputs. During development, you can attempt such verification earlier than testing since it's performed modularly on a per-subprogram basis. For example, in the code shown above, you can formally verify Use_Find even before you write the body for subprogram Find.

63.2 Advanced Contracts

Contracts for functional correctness are usually more complex than contracts for program integrity, so they more often require you to use the new forms of expressions introduced by the Ada 2012 standard. In particular, quantified expressions, which allow you to specify properties that must hold for all or for at least one element of a range, come in handy when specifying properties of arrays.

As contracts become more complex, you may find it useful to introduce new abstractions to improve the readability of your contracts. Expression functions are a good means to this end because you can retain their bodies in your package's specification.

Finally, some properties, especially those better described as invariants over data than as properties of subprograms, may be cumbersome to express as subprogram contracts. Type predicates, which must hold for every object of a given type, are usually a better match for this purpose. Here's an example.

Listing 1926: show_sort.ads

```

1 package Show_Sort is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Is_Sorted (A : Nat_Array) return Boolean is
6     (for all I in A'Range =>
7       (if I < A'Last then A (I) <= A (I + 1)));

```

(continues on next page)

(continued from previous page)

```

8  -- Returns True if A is sorted in increasing order.
9
10 subtype Sorted_Nat_Array is Nat_Array with
11     Dynamic_Predicate => Is_Sorted (Sorted_Nat_Array);
12 -- Elements of type Sorted_Nat_Array are all sorted.
13
14 Good_Array : Sorted_Nat_Array := (1, 2, 4, 8, 42);
15 end Show_Sort;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Sort
MD5: d3b3d26d62074d11b19d9282cc548c1b

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_sort.ads:5:13: info: implicit aspect Always_Terminates on "Is_Sorted" has
↳ been proved, subprogram will terminate
show_sort.ads:7:32: info: index check proved
show_sort.ads:7:43: info: overflow check proved
show_sort.ads:7:43: info: index check proved
show_sort.ads:14:37: info: range check proved
show_sort.ads:14:37: info: predicate check proved

```

We can use the subtype `Sorted_Nat_Array` as the type of a variable that must remain sorted throughout the program's execution. Specifying that an array is sorted requires a rather complex expression involving quantifiers, so we abstract away this property as an expression function to improve readability. `Is_Sorted`'s body remains in the package's specification and allows users of the package to retain a precise knowledge of its meaning when necessary. (You must use `Nat_Array` as the type of the operand of `Is_Sorted`. If you use `Sorted_Nat_Array`, you'll get infinite recursion at runtime when assertion checks are enabled since that function is called to check all operands of type `Sorted_Nat_Array`.)

63.2.1 Ghost Code

As the properties you need to specify grow more complex, you may have entities that are only needed because they are used in specifications (contracts). You may find it important to ensure that these entities can't affect the behavior of the program or that they're completely removed from production code. This concept, having entities that are only used for specifications, is usually called having *ghost* code and is supported in SPARK by the Ghost aspect.

You can use Ghost aspects to annotate any entity including variables, types, subprograms, and packages. If you mark an entity as Ghost, GNATprove ensures it can't affect the program's behavior. When the program is compiled with assertions enabled, ghost code is executed like normal code so it can execute the contracts using it. You can also instruct the compiler to not generate code for ghost entities.

Consider the procedure `Do_Something` below, which calls a complex function on its input, `X`, and wants to check that the initial and modified values of `X` are related in that complex way.

Listing 1927: show_ghost.ads

```

1 package Show_Ghost is
2

```

(continues on next page)

(continued from previous page)

```

3  type T is record
4      A, B, C, D, E : Boolean;
5  end record;
6
7  function Formula (X : T) return Boolean is
8      ((X.A and X.B) or (X.C and (X.D or X.E)));
9
10 function Is_Correct (X, Y : T) return Boolean is
11     (Formula (X) = Formula (Y));
12
13 procedure Do_Something (X : in out T);
14
15 end Show_Ghost;
```

Listing 1928: show_ghost.adb

```

1  package body Show_Ghost is
2
3      procedure Do_Some_Complex_Stuff (X : in out T) is
4          begin
5              X := T'(X.B, X.A, X.C, X.E, X.D);
6          end Do_Some_Complex_Stuff;
7
8      procedure Do_Something (X : in out T) is
9          X_Init : constant T := X with Ghost;
10         begin
11             Do_Some_Complex_Stuff (X);
12             pragma Assert (Is_Correct (X_Init, X));
13             -- It is OK to use X_Init inside an assertion.
14         end Do_Something;
15
16 end Show_Ghost;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_1
MD5: 0a6caaec950b3b043a53c18bab3cb39b

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_ghost.adb:12:22: info: assertion proved
show_ghost.ads:7:13: info: implicit aspect Always_Terminates on "Formula" has been_
↳proved, subprogram will terminate
show_ghost.ads:10:13: info: implicit aspect Always_Terminates on "Is_Correct" has_
↳been proved, subprogram will terminate
```

Do_Something stores the initial value of X in a ghost constant, X_Init. We reference it in an assertion to check that the computation performed by the call to Do_Some_Complex_Stuff modified the value of X in the expected manner.

However, X_Init can't be used in normal code, for example to restore the initial value of X.

Listing 1929: show_ghost.ads

```

1  package Show_Ghost is
2
3      type T is record
```

(continues on next page)

(continued from previous page)

```

4     A, B, C, D, E : Boolean;
5 end record;
6
7 function Formula (X : T) return Boolean is
8     ((X.A and X.B) or (X.C and (X.D or X.E)));
9
10 function Is_Correct (X, Y : T) return Boolean is
11     (Formula (X) = Formula (Y));
12
13 procedure Do_Something (X : in out T);
14
15 end Show_Ghost;
```

Listing 1930: show_ghost.adb

```

1 package body Show_Ghost is
2
3     procedure Do_Some_Complex_Stuff (X : in out T) is
4     begin
5         X := T'(X.B, X.A, X.C, X.E, X.D);
6     end Do_Some_Complex_Stuff;
7
8     procedure Do_Something (X : in out T) is
9     X_Init : constant T := X with Ghost;
10    begin
11        Do_Some_Complex_Stuff (X);
12        pragma Assert (Is_Correct (X_Init, X));
13
14        X := X_Init; -- ERROR
15
16    end Do_Something;
17
18 end Show_Ghost;
```

Listing 1931: use_ghost.adb

```

1 with Show_Ghost; use Show_Ghost;
2
3 procedure Use_Ghost is
4     X : T := (True, True, False, False, True);
5 begin
6     Do_Something (X);
7 end Use_Ghost;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_2
MD5: 464bb4bc355a648e2b92940ec80b4717

Build output

```
show_ghost.adb:14:12: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed
```

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
```

(continues on next page)

(continued from previous page)

```
Phase 2 of 3: generation of Global contracts ...
show_ghost.adb:14:12: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts
```

When compiling this example, the compiler flags the use of `X_Init` as illegal, but more complex cases of interference between ghost and normal code may sometimes only be detected when you run GNATprove.

63.2.2 Ghost Functions

Functions used only in specifications are a common occurrence when writing contracts for functional correctness. For example, expression functions used to simplify or factor out common patterns in contracts can usually be marked as ghost.

But ghost functions can do more than improve readability. In real-world programs, it's often the case that some information necessary for functional specification isn't accessible in the package's specification because of abstraction.

Making this information available to users of the packages is generally out of the question because that breaks the abstraction. Ghost functions come in handy in that case since they provide a way to give access to that information without making it available to normal client code.

Let's look at the following example.

Listing 1932: stacks.ads

```

1 package Stacks is
2
3   pragma Unevaluated_Use_Of_Old (Allow);
4
5   type Stack is private;
6
7   type Element is new Natural;
8   type Element_Array is array (Positive range <>) of Element;
9   Max : constant Natural := 100;
10
11  function Get_Model (S : Stack) return Element_Array with Ghost;
12  -- Returns an array as a model of a stack.
13
14  procedure Push (S : in out Stack; E : Element) with
15    Pre => Get_Model (S)'Length < Max,
16    Post => Get_Model (S) = Get_Model (S)'Old & E;
17
18 private
19
20   subtype Length_Type is Natural range 0 .. Max;
21
22   type Stack is record
23     Top      : Length_Type := 0;
24     Content : Element_Array (1 .. Max) := (others => 0);
25   end record;
26
27 end Stacks;
```

Code block metadata

```
Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_Functions
MD5: e287612bd66753f07ac3eecb36c693de
```

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

```

Here, the type `Stack` is private. To specify the expected behavior of the `Push` procedure, we need to go inside this abstraction and access the values of the elements stored in `S`. For this, we introduce a function `Get_Model` that returns an array as a representation of the stack. However, we don't want code that uses the `Stack` package to use `Get_Model` in normal code since this breaks our stack's abstraction.

Here's an example of trying to break that abstraction in the subprogram `Peek` below.

Listing 1933: stacks.ads

```

1 package Stacks is
2
3   pragma Unevaluated_Use_Of_Old (Allow);
4
5   type Stack is private;
6
7   type Element is new Natural;
8   type Element_Array is array (Positive range <>) of Element;
9   Max : constant Natural := 100;
10
11  function Get_Model (S : Stack) return Element_Array with Ghost;
12  -- Returns an array as a model of a stack.
13
14  procedure Push (S : in out Stack; E : Element) with
15    Pre => Get_Model (S)'Length < Max,
16    Post => Get_Model (S) = Get_Model (S)'Old & E;
17
18  function Peek (S : Stack; I : Positive) return Element is
19    (Get_Model (S) (I)); -- ERROR
20
21 private
22
23   subtype Length_Type is Natural range 0 .. Max;
24
25   type Stack is record
26     Top      : Length_Type := 0;
27     Content : Element_Array (1 .. Max) := (others => 0);
28   end record;
29
30 end Stacks;

```

Code block metadata

```

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_Model
MD5: c00b5d86c9d0b665ccdda7f68f16f07a

```

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
stacks.ads:19:07: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

```

We see that marking the function as Ghost achieves this goal: it ensures that the subprogram `Get_Model` is never used in production code.

63.2.3 Global Ghost Variables

Though it happens less frequently, you may have specifications requiring you to store additional information in global variables that isn't needed in normal code. You should mark these global variables as ghost, allowing the compiler to remove them when assertions aren't enabled. You can use these variables for any purpose within the contracts that make up your specifications. A common scenario is writing specifications for subprograms that modify a complex or private global data structure: you can use these variables to provide a model for that structure that's updated by the ghost code as the program modifies the data structure itself.

You can also use ghost variables to store information about previous runs of subprograms to specify temporal properties. In the following example, we have two procedures, one that accesses a state A and the other that accesses a state B. We use the ghost variable `Last_Accessed_Is_A` to specify that B can't be accessed twice in a row without accessing A in between.

Listing 1934: `call_sequence.ads`

```
1 package Call_Sequence is
2
3   type T is new Integer;
4
5   Last_Accessed_Is_A : Boolean := False with Ghost;
6
7   procedure Access_A with
8     Post => Last_Accessed_Is_A;
9
10  procedure Access_B with
11    Pre  => Last_Accessed_Is_A,
12    Post => not Last_Accessed_Is_A;
13    -- B can only be accessed after A
14
15 end Call_Sequence;
```

Listing 1935: `call_sequence.adb`

```
1 package body Call_Sequence is
2
3   procedure Access_A is
4     begin
5       -- ...
6       Last_Accessed_Is_A := True;
7     end Access_A;
8
9   procedure Access_B is
10    begin
11      -- ...
12      Last_Accessed_Is_A := False;
13    end Access_B;
14
15 end Call_Sequence;
```

Listing 1936: `main.adb`

```
1 with Call_Sequence; use Call_Sequence;
2
```

(continues on next page)

(continued from previous page)

```

3 procedure Main is
4 begin
5     Access_A;
6     Access_B;
7     Access_B; -- ERROR
8 end Main;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Global_Ghost_Vars
MD5: f33fa2ad2bd31eb03d4400c78f22eb71

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
main.adb:7:04: medium: precondition might fail
gnatprove: unproved check messages considered as errors

Runtime output

```

raised ADA.ASSERTIONS.ASSERTION_ERROR : failed precondition from call_sequence.
↪ads:11

```

Let's look at another example. The specification of a subprogram's expected behavior is sometimes best expressed as a sequence of actions it must perform. You can use global ghost variables that store intermediate values of normal variables to write this sort of specification more easily.

For example, we specify the subprogram `Do_Two_Things` below in two steps, using the ghost variable `V_Interm` to store the intermediate value of `V` between those steps. We could also express this using an existential quantification on the variable `V_Interm`, but it would be impractical to iterate over all integers at runtime and this can't always be written in SPARK because quantification is restricted to `for ... loop` patterns.

Finally, supplying the value of the variable may help the prover verify the contracts.

Listing 1937: action_sequence.ads

```

1 package Action_Sequence is
2
3     type T is new Integer;
4
5     V_Interm : T with Ghost;
6
7     function First_Thing_Done (X, Y : T) return Boolean with Ghost;
8     function Second_Thing_Done (X, Y : T) return Boolean with Ghost;
9
10    procedure Do_Two_Things (V : in out T) with
11        Post => First_Thing_Done (V'Old, V_Interm)
12        and then Second_Thing_Done (V_Interm, V);
13
14 end Action_Sequence;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Intermediate_Values
MD5: 2ffbd2cb187c0a81423c78e0989d62f0

Prover output

```
Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
```

Note

For more details on ghost code, see the [SPARK User's Guide](#)⁴⁰⁸.

63.3 Guide Proof

Since properties of interest for functional correctness are more complex than those involved in proofs of program integrity, we expect GNATprove to initially be unable to verify them even though they're valid. You'll find the techniques we discussed in [Debugging Failed Proof Attempts](#) (page 1467) to come in handy here. We now go beyond those techniques and focus on more ways of improving results in the cases where the property is valid but GNATprove can't prove it in a reasonable amount of time.

In those cases, you may want to try and guide GNATprove to either complete the proof or strip it down to a small number of easily-reviewable assumptions. For this purpose, you can add assertions to break complex proofs into smaller steps.

```
pragma Assert (Assertion_Checked_By_The_Tool);
-- info: assertion proved

pragma Assert (Assumption_Validated_By_Other_Means);
-- medium: assertion might fail

pragma Assume (Assumption_Validated_By_Other_Means);
-- The tool does not attempt to check this expression.
-- It is recorded as an assumption.
```

One such intermediate step you may find useful is to try to prove a theoretically-equivalent version of the desired property, but one where you've simplified things for the prover, such as by splitting up different cases or inlining the definitions of functions.

Some intermediate assertions may not be proved by GNATprove either because it's missing some information or because the amount of information available is confusing. You can verify these remaining assertions by other means such as testing (since they're executable) or by review. You can then choose to instruct GNATprove to ignore them, either by turning them into assumptions, as in our example, or by using a **pragma Annotate**. In both cases, the compiler generates code to check these assumptions at runtime when you enable assertions.

63.3.1 Local Ghost Variables

You can use ghost code to enhance what you can express inside intermediate assertions in the same way we did above to enhance our contracts in specifications. In particular, you'll commonly have local variables or constants whose only purpose is to be used in assertions. You'll mostly use these ghost variables to store previous values of variables or expressions you want to refer to in assertions. They're especially useful to refer to initial values of parameters and expressions since the `'old` attribute is only allowed in postconditions.

⁴⁰⁸ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/specification_features.html#ghost-code

In the example below, we want to help GNATprove verify the postcondition of P. We do this by introducing a local ghost constant, X_Init, to represent this value and writing an assertion in both branches of an **if** statement that repeats the postcondition, but using X_Init.

Listing 1938: show_local_ghost.ads

```

1 package Show_Local_Ghost is
2
3     type T is new Natural;
4
5     function F (X, Y : T) return Boolean is (X > Y) with Ghost;
6
7     function Condition (X : T) return Boolean is (X mod 2 = 0);
8
9     procedure P (X : in out T) with
10         Pre  => X < 1_000_000,
11         Post => F (X, X'Old);
12
13 end Show_Local_Ghost;
```

Listing 1939: show_local_ghost.adb

```

1 package body Show_Local_Ghost is
2
3     procedure P (X : in out T) is
4         X_Init : constant T := X with Ghost;
5     begin
6         if Condition (X) then
7             X := X + 1;
8             pragma Assert (F (X, X_Init));
9         else
10            X := X * 2;
11            pragma Assert (F (X, X_Init));
12        end if;
13    end P;
14
15 end Show_Local_Ghost;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Local_Ghost
MD5: 071ee53a06a6b5880eee6e9ea06dbcf3

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_local_ghost.adb:7:17: info: overflow check proved
show_local_ghost.adb:8:25: info: assertion proved
show_local_ghost.adb:10:17: info: overflow check proved
show_local_ghost.adb:11:25: info: assertion proved
show_local_ghost.ads:5:13: info: implicit aspect Always_Terminates on "F" has been_
↳proved, subprogram will terminate
show_local_ghost.ads:7:13: info: implicit aspect Always_Terminates on "Condition"
↳has been proved, subprogram will terminate
show_local_ghost.ads:7:52: info: division check proved
show_local_ghost.ads:11:14: info: postcondition proved
```

You can also use local ghost variables for more complex purposes such as building a data structure that serves as witness for a complex property of a subprogram. In our example,

we want to prove that the Sort procedure doesn't create new elements, that is, that all the elements present in A after the sort were in A before the sort. This property isn't enough to ensure that a call to Sort produces a value for A that's a permutation of its value before the call (or that the values are indeed sorted). However, it's already complex for a prover to verify because it involves a nesting of quantifiers. To help GNATprove, you may find it useful to store, for each index I, an index J that has the expected property.

```

procedure Sort (A : in out Nat_Array) with
  Post => (for all I in A'Range =>
    (for some J in A'Range => A (I) = A'Old (J)))
is
  Permutation : Index_Array := (1 => 1, 2 => 2, ...) with Ghost;
begin
  ...
end Sort;

```

63.3.2 Ghost Procedures

Ghost procedures can't affect the value of normal variables, so they're mostly used to perform operations on ghost variables or to group together a set of intermediate assertions.

Abstracting away the treatment of assertions and ghost variables inside a ghost procedure has several advantages. First, you're allowed to use these variables in any way you choose in code inside ghost procedures. This isn't the case outside ghost procedures, where the only ghost statements allowed are assignments to ghost variables and calls to ghost procedures.

As an example, the **for** loop contained in Increase_A couldn't appear by itself in normal code.

Listing 1940: show_ghost_proc.ads

```

1 package Show_Ghost_Proc is
2
3   type Nat_Array is array (Integer range <>) of Natural;
4
5   A : Nat_Array (1 .. 100) with Ghost;
6
7   procedure Increase_A with
8     Ghost,
9     Pre => (for all I in A'Range => A (I) < Natural'Last);
10
11 end Show_Ghost_Proc;

```

Listing 1941: show_ghost_proc.adb

```

1 package body Show_Ghost_Proc is
2
3   procedure Increase_A is
4   begin
5     for I in A'Range loop
6       A (I) := A (I) + 1;
7     end loop;
8   end Increase_A;
9
10 end Show_Ghost_Proc;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Ghost_Proc
 MD5: 4b9cfe25011169a0cd3b4a3b03135dc4

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_ghost_proc.adb:6:25: info: overflow check proved
```

Using the abstraction also improves readability by hiding complex code that isn't part of the functional behavior of the subprogram. Finally, it can help GNATprove by abstracting away assertions that would otherwise make its job more complex.

In the example below, calling `Prove_P` with `X` as an operand only adds `P (X)` to the proof context instead of the larger set of assertions required to verify it. In addition, the proof of `P` need only be done once and may be made easier not having any unnecessary information present in its context while verifying it. Also, if GNATprove can't fully verify `Prove_P`, you can review the remaining assumptions more easily since they're in a smaller context.

```
procedure Prove_P (X : T) with Ghost,
  Global => null,
  Post   => P (X);
```

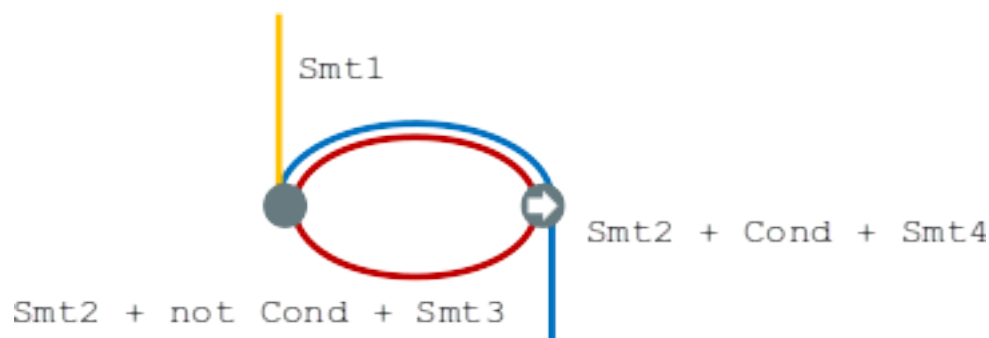
63.3.3 Handling of Loops

When the program involves a loop, you're almost always required to provide additional annotations to allow GNATprove to complete a proof because the verification techniques used by GNATprove don't handle cycles in a subprogram's control flow. Instead, loops are flattened by dividing them into several acyclic parts.

As an example, let's look at a simple loop with an exit condition.

```
Smt1;
loop
  Smt2;
  exit when Cond;
  Smt3;
end loop;
Smt4;
```

As shown below, the control flow is divided into three parts.



The first, shown in yellow, starts earlier in the subprogram and enters the loop statement. The loop itself is divided into two parts. Red represents a complete execution of the loop's body: an execution where the exit condition isn't satisfied. Blue represents the last execution of the loop, which includes some of the subprogram following it. For that path, the exit condition is assumed to hold. The red and blue parts are always executed after the yellow one.

GNATprove analyzes these parts independently since it doesn't have a way to track how variables may have been updated by an iteration of the loop. It forgets everything it knows

about those variables from one part when entering another part. However, values of constants and variables that aren't modified in the loop are not an issue.

In other words, handling loops in that way makes GNATprove imprecise when verifying a subprogram involving a loop: it can't verify a property that relies on values of variables modified inside the loop. It won't forget any information it had on the value of constants or unmodified variables, but it nevertheless won't be able to deduce new information about them from the loop.

For example, consider the function Find which iterates over the array A and searches for an element where E is stored in A.

Listing 1942: show_find.ads

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;
```

Listing 1943: show_find.adb

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             pragma Assert (for all J in A'First .. I - 1 => A (J) /= E);
7             -- assertion is not proved
8             if A (I) = E then
9                 return I;
10            end if;
11            pragma Assert (A (I) /= E);
12            -- assertion is proved
13        end loop;
14        return 0;
15    end Find;
16
17 end Show_Find;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop
MD5: cb9cd0cb102c3baba3b21a788b6e4ae3

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_find.adb:6:51: info: overflow check proved
show_find.adb:6:58: medium: assertion might fail, cannot prove A (J) /= E
↳[possible fix: subprogram at show_find.ads:5 should mention A and E in a
↳precondition]
show_find.adb:6:61: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:11:25: info: assertion proved
show_find.ads:5:13: info: implicit aspect Always_Terminates on "Find" has been
↳proved, subprogram will terminate
gnatprove: unproved check messages considered as errors
```

At the end of each loop iteration, GNATprove knows that the value stored at index *I* in *A* must not be *E*. (If it were, the loop wouldn't have reached the end of the iteration.) This proves the second assertion. But it's unable to aggregate this information over multiple loop iterations to deduce that it's true for all the indexes smaller than *I*, so it can't prove the first assertion.

63.3.4 Loop Invariants

To overcome these limitations, you can provide additional information to GNATprove in the form of a *loop invariant*. In SPARK, a loop invariant is a Boolean expression which holds true at every iteration of the loop. Like other assertions, you can have it checked at runtime by compiling the program with assertions enabled.

The major difference between loop invariants and other assertions is the way it's treated for proofs. GNATprove performs the proof of a loop invariant in two steps: first, it checks that it holds for the first iteration of the loop and then it checks that it holds in an arbitrary iteration assuming it held in the previous iteration. This is called *proof by induction*⁴⁰⁹.

As an example, let's add a loop invariant to the *Find* function stating that the first element of *A* is not *E*.

Listing 1944: show_find.ads

```

1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;
```

Listing 1945: show_find.adb

```

1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             pragma Loop_Invariant (A (A'First) /= E);
7             -- loop invariant not proved in first iteration
8             -- but preservation of loop invariant is proved
9             if A (I) = E then
10                 return I;
11             end if;
12         end loop;
13         return 0;
14     end Find;
15
16 end Show_Find;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_1
MD5: 8d5fefdc9deacd4eb50850be91fbefe

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...

(continues on next page)

⁴⁰⁹ https://en.wikipedia.org/wiki/Mathematical_induction

(continued from previous page)

```
Phase 3 of 3: flow analysis and proof ...
show_find.adb:6:33: medium: loop invariant might fail in first iteration [possible_
↳fix: subprogram at show_find.ads:5 should mention A and E in a precondition]
show_find.adb:6:33: info: loop invariant preservation proved
show_find.adb:6:37: info: index check proved
show_find.adb:10:20: info: range check proved
show_find.ads:5:13: info: implicit aspect Always_Terminates on "Find" has been_
↳proved, subprogram will terminate
gnatprove: unproved check messages considered as errors
```

To verify this invariant, GNATprove generates two checks. The first checks that the assertion holds in the first iteration of the loop. This isn't verified by GNATprove. And indeed there's no reason to expect the first element of A to always be different from E in this iteration. However, the second check is proved: it's easy to deduce that if the first element of A was not E in a given iteration it's still not E in the next. However, if we move the invariant to the end of the loop, then it is successfully verified by GNATprove.

Not only do loop invariants allow you to verify complex properties of loops, but GNATprove also uses them to verify other properties, such as the absence of runtime errors over both the loop's body and the statements following the loop. More precisely, when verifying a runtime check or other assertion there, GNATprove assumes that the last occurrence of the loop invariant preceding the check or assertion is true.

Let's look at a version of Find where we use a loop invariant instead of an assertion to state that none of the array elements seen so far are equal to E.

Listing 1946: show_find.ads

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;
```

Listing 1947: show_find.adb

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5         for I in A'Range loop
6             pragma Loop_Invariant
7                 (for all J in A'First .. I - 1 => A (J) /= E);
8             if A (I) = E then
9                 return I;
10            end if;
11        end loop;
12        pragma Assert (for all I in A'Range => A (I) /= E);
13        return 0;
14    end Find;
15
16 end Show_Find;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_2
MD5: 21588161eaddb82f54c3cb3dcc14a6ac

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_find.adb:7:13: info: loop invariant initialization proved
show_find.adb:7:13: info: loop invariant preservation proved
show_find.adb:7:39: info: overflow check proved
show_find.adb:7:49: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:12:22: info: assertion proved
show_find.adb:12:49: info: index check proved
show_find.ads:5:13: info: implicit aspect Always_Terminates on "Find" has been
→proved, subprogram will terminate

```

This version is fully verified by GNATprove! This time, it proves that the loop invariant holds in every iteration of the loop (separately proving this property for the first iteration and then for the following iterations). It also proves that none of the elements of A are equal to E after the loop exits by assuming that the loop invariant holds in the last iteration of the loop.

Note

For more details on loop invariants, see the [SPARK User's Guide](#)⁴¹⁰.

Finding a good loop invariant can turn out to be quite a challenge. To make this task easier, let's review the four good properties of a good loop invariant:

Prop-erty	Description
INIT	It should be provable in the first iteration of the loop.
INSIDE	It should allow proving the absence of run-time errors and local assertions inside the loop.
AFTER	It should allow proving absence of run-time errors, local assertions, and the subprogram postcondition after the loop.
PRE-SERVE	It should be provable after the first iteration of the loop.

Let's look at each of these in turn. First, the loop invariant should be provable in the first iteration of the loop (INIT). If your invariant fails to achieve this property, you can debug the loop invariant's initialization like any failing proof attempt using strategies for [Debugging Failed Proof Attempts](#) (page 1467).

Second, the loop invariant should be precise enough to allow GNATprove to prove absence of runtime errors in both statements from the loop's body (INSIDE) and those following the loop (AFTER). To do this, you should remember that all information concerning a variable modified in the loop that's not included in the invariant is forgotten by GNATprove. In particular, you should take care to include in your invariant what's usually called the loop's *frame condition*, which lists properties of variables that are true throughout the execution of the loop even though those variables are modified by the loop.

Finally, the loop invariant should be precise enough to prove that it's preserved through successive iterations of the loop (PRESERVE). This is generally the trickiest part. To understand why GNATprove hasn't been able to verify the preservation of a loop invariant you provided, you may find it useful to repeat it as local assertions throughout the loop's body to determine at which point it can no longer be proved.

⁴¹⁰ https://docs.adacore.com/live/wave/spark2014/html/spark2014 Ug/en/source/assertion_pragmas.html#loop-invariants

As an example, let's look at a loop that iterates through an array A and applies a function F to each of its elements.

Listing 1948: show_map.ads

```
1 package Show_Map is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function F (V : Natural) return Natural is
6         (if V /= Natural'Last then V + 1 else V);
7
8     procedure Map (A : in out Nat_Array);
9
10 end Show_Map;
```

Listing 1949: show_map.adb

```
1 package body Show_Map is
2
3     procedure Map (A : in out Nat_Array) is
4         A_I : constant Nat_Array := A with Ghost;
5     begin
6         for K in A'Range loop
7             A (K) := F (A (K));
8             pragma Loop_Invariant
9                 (for all J in A'First .. K => A (J) = F (A'Loop_Entry (J)));
10        end loop;
11        pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
12    end Map;
13
14 end Show_Map;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_3
MD5: 1a4583c9b2b772f79bcf29cff0caa96a

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_map.adb:9:13: info: loop invariant initialization proved
show_map.adb:9:13: info: loop invariant preservation proved
show_map.adb:9:45: info: index check proved
show_map.adb:9:67: info: index check proved
show_map.adb:11:22: info: assertion proved
show_map.adb:11:49: info: index check proved
show_map.adb:11:62: info: index check proved
show_map.ads:5:13: info: implicit aspect Always_Terminates on "F" has been proved,
↳subprogram will terminate
show_map.ads:6:35: info: overflow check proved
```

After the loop, each element of A should be the result of applying F to its previous value. We want to prove this. To specify this property, we copy the value of A before the loop into a ghost variable, A_I. Our loop invariant states that the element at each index less than K has been modified in the expected way. We use the Loop_Entry attribute to refer to the value of A on entry of the loop instead of using A_I.

Does our loop invariant have the four properties of a good loop-invariant? When launching GNATprove, we see that INIT is fulfilled: the invariant's initialization is proved. So are INSIDE and AFTER: no potential runtime errors are reported and the assertion following the

loop is successfully verified.

The situation is slightly more complex for the PRESERVE property. GNATprove manages to prove that the invariant holds after the first iteration thanks to the automatic generation of frame conditions. It was able to do this because it completes the provided loop invariant with the following frame condition stating what part of the array hasn't been modified so far:

```
pragma Loop_Invariant
  (for all J in K .. A'Last => A (J) = (if J > K then A'Loop_Entry (J))));
```

GNATprove then uses both our and the internally-generated loop invariants to prove PRESERVE. However, in more complex cases, the heuristics used by GNATprove to generate the frame condition may not be sufficient and you'll have to provide one as a loop invariant. For example, consider a version of Map where the result of applying F to an element at index K is stored at index K-1:

Listing 1950: show_map.ads

```
1 package Show_Map is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function F (V : Natural) return Natural is
6     (if V /= Natural'Last then V + 1 else V);
7
8   procedure Map (A : in out Nat_Array);
9
10 end Show_Map;
```

Listing 1951: show_map.adb

```
1 package body Show_Map is
2
3   procedure Map (A : in out Nat_Array) is
4     A_I : constant Nat_Array := A with Ghost;
5   begin
6     for K in A'Range loop
7       if K /= A'First then
8         A (K - 1) := F (A (K));
9       end if;
10      pragma Loop_Invariant
11        (for all J in A'First .. K =>
12          (if J /= A'First then A (J - 1) = F (A'Loop_Entry (J))));
13      -- pragma Loop_Invariant
14      -- (for all J in K .. A'Last => A (J) = A'Loop_Entry (J));
15    end loop;
16    pragma Assert (for all K in A'Range =>
17      (if K /= A'First then A (K - 1) = F (A_I (K))));
18  end Map;
19
20 end Show_Map;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Loop_Invariant_4
MD5: 6c51768547d3baa2c19d0e33959388fe

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...

(continues on next page)

(continued from previous page)

```

Phase 3 of 3: flow analysis and proof ...
show_map.adb:8:18: info: overflow check proved
show_map.adb:8:18: info: index check proved
show_map.adb:11:13: info: loop invariant initialization proved
show_map.adb:12:36: medium: loop invariant might not be preserved by an arbitrary
↳ iteration, cannot prove A (J - 1) = F (A'Loop_Entry (J))
show_map.adb:12:41: info: overflow check proved
show_map.adb:12:41: info: index check proved
show_map.adb:12:65: info: index check proved
show_map.adb:16:22: info: assertion proved
show_map.adb:17:50: info: overflow check proved
show_map.adb:17:50: info: index check proved
show_map.adb:17:65: info: index check proved
show_map.ads:5:13: info: implicit aspect Always_Terminates on "F" has been proved,
↳ subprogram will terminate
show_map.ads:6:35: info: overflow check proved
gnatprove: unproved check messages considered as errors

```

You need to uncomment the second loop invariant containing the frame condition in order to prove the assertion after the loop.

Note

For more details on how to write a loop invariant, see the [SPARK User's Guide](#)⁴¹¹.

63.4 Code Examples / Pitfalls

This section contains some code examples and pitfalls.

63.4.1 Example #1

We implement a ring buffer inside an array `Content`, where the contents of a ring buffer of length `Length` are obtained by starting at index `First` and possibly wrapping around the end of the buffer. We use a ghost function `Get_Model` to return the contents of the ring buffer for use in contracts.

Listing 1952: ring_buffer.ads

```

1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4
5   type Nat_Array is array (Positive range <>) of Natural;
6
7   function Get_Model return Nat_Array with Ghost;
8
9   procedure Push_Last (E : Natural) with
10     Pre  => Get_Model'Length < Max_Size,
11     Post => Get_Model'Length = Get_Model'Old'Length + 1;
12
13 end Ring_Buffer;

```

⁴¹¹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_loop_invariants.html

Listing 1953: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3     subtype Length_Range is Natural range 0 .. Max_Size;
4     subtype Index_Range  is Natural range 1 .. Max_Size;
5
6     Content : Nat_Array (1 .. Max_Size) := (others => 0);
7     First   : Index_Range                := 1;
8     Length  : Length_Range               := 0;
9
10    function Get_Model return Nat_Array with
11        Refined_Post => Get_Model'Result'Length = Length
12    is
13        Size : constant Length_Range := Length;
14        Result : Nat_Array (1 .. Size) := (others => 0);
15    begin
16        if First + Length - 1 <= Max_Size then
17            Result := Content (First .. First + Length - 1);
18        else
19            declare
20                Len : constant Length_Range := Max_Size - First + 1;
21            begin
22                Result (1 .. Len) := Content (First .. Max_Size);
23                Result (Len + 1 .. Length) := Content (1 .. Length - Len);
24            end;
25        end if;
26        return Result;
27    end Get_Model;
28
29    procedure Push_Last (E : Natural) is
30    begin
31        if First + Length <= Max_Size then
32            Content (First + Length) := E;
33        else
34            Content (Length - Max_Size + First) := E;
35        end if;
36        Length := Length + 1;
37    end Push_Last;
38
39 end Ring_Buffer;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_01
MD5: 3afd7d58f97001618acc05062115f1a3

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
ring_buffer.adb:11:22: info: refined post proved
ring_buffer.adb:11:38: info: range check proved
ring_buffer.adb:14:07: info: range check proved
ring_buffer.adb:14:41: info: length check proved
ring_buffer.adb:17:17: info: length check proved
ring_buffer.adb:17:20: info: range check proved
ring_buffer.adb:17:20: info: length check proved
ring_buffer.adb:20:61: info: range check proved
ring_buffer.adb:22:13: info: range check proved

```

(continues on next page)

(continued from previous page)

```

ring_buffer.adb:22:21: info: index check proved
ring_buffer.adb:22:26: info: index check proved
ring_buffer.adb:22:31: info: length check proved
ring_buffer.adb:22:34: info: range check proved
ring_buffer.adb:22:34: info: length check proved
ring_buffer.adb:23:13: info: range check proved
ring_buffer.adb:23:25: info: index check proved
ring_buffer.adb:23:32: info: index check proved
ring_buffer.adb:23:40: info: length check proved
ring_buffer.adb:23:43: info: range check proved
ring_buffer.adb:23:43: info: length check proved
ring_buffer.adb:32:25: info: index check proved
ring_buffer.adb:34:37: info: index check proved
ring_buffer.adb:36:24: info: range check proved
ring_buffer.ads:7:13: info: implicit aspect Always_Terminates on "Get_Model" has_
been proved, subprogram will terminate
ring_buffer.ads:11:14: info: postcondition proved

```

This is correct: `Get_Model` is used only in contracts. Calls to `Get_Model` make copies of the buffer's contents, which isn't efficient, but is fine because `Get_Model` is only used for verification, not in production code. We enforce this by making it a ghost function. We'll produce the final production code with appropriate compiler switches (i.e., not using `-gnata`) that ensure assertions are ignored.

63.4.2 Example #2

Instead of using a ghost function, `Get_Model`, to retrieve the contents of the ring buffer, we're now using a global ghost variable, `Model`.

Listing 1954: ring_buffer.ads

```

1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4   subtype Length_Range is Natural range 0 .. Max_Size;
5   subtype Index_Range is Natural range 1 .. Max_Size;
6
7   type Nat_Array is array (Positive range <>) of Natural;
8
9   type Model_Type (Length : Length_Range := 0) is record
10     Content : Nat_Array (1 .. Length);
11   end record
12     with Ghost;
13
14   Model : Model_Type with Ghost;
15
16   function Valid_Model return Boolean;
17
18   procedure Push_Last (E : Natural) with
19     Pre  => Valid_Model
20     and then Model.Length < Max_Size,
21     Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;

```

Listing 1955: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);

```

(continues on next page)

(continued from previous page)

```

4   First   : Index_Range           := 1;
5   Length  : Length_Range          := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is
11  begin
12    if First + Length <= Max_Size then
13      Content (First + Length) := E;
14    else
15      Content (Length - Max_Size + First) := E;
16    end if;
17    Length := Length + 1;
18  end Push_Last;
19
20 end Ring_Buffer;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_02
MD5: 144f58bd95cd460e4ed388d4f3351fe3

Build output

```

ring_buffer.adb:8:08: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed

```

Prover output

```

Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
ring_buffer.adb:8:08: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

```

This example isn't correct. `Model`, which is a ghost variable, must not influence the return value of the normal function `Valid_Model`. Since `Valid_Model` is only used in specifications, we should have marked it as `Ghost`. Another problem is that `Model` needs to be updated inside `Push_Last` to reflect the changes to the ring buffer.

63.4.3 Example #3

Let's mark `Valid_Model` as `Ghost` and update `Model` inside `Push_Last`.

Listing 1956: ring_buffer.ads

```

1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4   subtype Length_Range is Natural range 0 .. Max_Size;
5   subtype Index_Range is Natural range 1 .. Max_Size;
6
7   type Nat_Array is array (Positive range <>) of Natural;
8
9   type Model_Type (Length : Length_Range := 0) is record
10     Content : Nat_Array (1 .. Length);
11 end record

```

(continues on next page)

(continued from previous page)

```

12     with Ghost;
13
14     Model : Model_Type with Ghost;
15
16     function Valid_Model return Boolean with Ghost;
17
18     procedure Push_Last (E : Natural) with
19         Pre => Valid_Model
20         and then Model.Length < Max_Size,
21         Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;

```

Listing 1957: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3     Content : Nat_Array (1 .. Max_Size) := (others => 0);
4     First   : Index_Range := 1;
5     Length  : Length_Range := 0;
6
7     function Valid_Model return Boolean is
8         (Model.Content'Length = Length);
9
10    procedure Push_Last (E : Natural) is
11    begin
12        if First + Length <= Max_Size then
13            Content (First + Length) := E;
14        else
15            Content (Length - Max_Size + First) := E;
16        end if;
17        Length := Length + 1;
18        Model := (Length => Model.Length + 1,
19                 Content => Model.Content & E);
20    end Push_Last;
21
22 end Ring_Buffer;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_03
MD5: 08b74f5fe560d238550a06c6323959cf

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
ring_buffer.adb:8:21: info: range check proved
ring_buffer.adb:13:25: info: index check proved
ring_buffer.adb:15:37: info: index check proved
ring_buffer.adb:17:24: info: range check proved
ring_buffer.adb:18:13: info: discriminant check proved
ring_buffer.adb:18:41: info: range check proved
ring_buffer.adb:19:42: info: range check proved
ring_buffer.adb:19:42: info: length check proved
ring_buffer.ads:10:07: info: range check proved
ring_buffer.ads:16:13: info: implicit aspect Always_Terminates on "Valid_Model"
↳ has been proved, subprogram will terminate
ring_buffer.ads:21:14: info: postcondition proved

```

This example is correct. The ghost variable `Model` can be referenced both from the body of the ghost function `Valid_Model` and the non-ghost procedure `Push_Last` as long as it's only used in ghost statements.

63.4.4 Example #4

We're now modifying `Push_Last` to share the computation of the new length between the operational and ghost code.

Listing 1958: ring_buffer.ads

```

1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4   subtype Length_Range is Natural range 0 .. Max_Size;
5   subtype Index_Range is Natural range 1 .. Max_Size;
6
7   type Nat_Array is array (Positive range <>) of Natural;
8
9   type Model_Type (Length : Length_Range := 0) is record
10     Content : Nat_Array (1 .. Length);
11   end record
12     with Ghost;
13
14   Model : Model_Type with Ghost;
15
16   function Valid_Model return Boolean with Ghost;
17
18   procedure Push_Last (E : Natural) with
19     Pre  => Valid_Model
20     and then Model.Length < Max_Size,
21     Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;
```

Listing 1959: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);
4   First   : Index_Range               := 1;
5   Length  : Length_Range              := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is
11    New_Length : constant Length_Range := Model.Length + 1;
12  begin
13    if First + Length <= Max_Size then
14      Content (First + Length) := E;
15    else
16      Content (Length - Max_Size + First) := E;
17    end if;
18    Length := New_Length;
19    Model := (Length => New_Length,
20             Content => Model.Content & E);
21  end Push_Last;
22
23 end Ring_Buffer;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_04
MD5: e27f0b4729be72d83f2cb981b1d00412

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
ring_buffer.adb:11:45: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

This example isn't correct. We didn't mark local constant `New_Length` as `Ghost`, so it can't be computed from the value of ghost variable `Model`. If we made `New_Length` a ghost constant, the compiler would report the problem on the assignment from `New_Length` to `Length`. The correct solution here is to compute `New_Length` from the value of the non-ghost variable `Length`.

63.4.5 Example #5

Let's move the code updating `Model` inside a local ghost procedure, `Update_Model`, but still using a local variable, `New_Length`, to compute the length.

Listing 1960: ring_buffer.ads

```
1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4   subtype Length_Range is Natural range 0 .. Max_Size;
5   subtype Index_Range is Natural range 1 .. Max_Size;
6
7   type Nat_Array is array (Positive range <>) of Natural;
8
9   type Model_Type (Length : Length_Range := 0) is record
10     Content : Nat_Array (1 .. Length);
11   end record
12     with Ghost;
13
14   Model : Model_Type with Ghost;
15
16   function Valid_Model return Boolean with Ghost;
17
18   procedure Push_Last (E : Natural) with
19     Pre => Valid_Model
20     and then Model.Length < Max_Size,
21     Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;
```

Listing 1961: ring_buffer.adb

```
1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);
4   First : Index_Range := 1;
5   Length : Length_Range := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is
11
```

(continues on next page)

(continued from previous page)

```

12   procedure Update_Model with Ghost is
13     New_Length : constant Length_Range := Model.Length + 1;
14   begin
15     Model := (Length => New_Length,
16              Content => Model.Content & E);
17   end Update_Model;
18
19   begin
20     if First + Length <= Max_Size then
21       Content (First + Length) := E;
22     else
23       Content (Length - Max_Size + First) := E;
24     end if;
25     Length := Length + 1;
26     Update_Model;
27   end Push_Last;
28
29 end Ring_Buffer;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_05
MD5: cc97fb35205c9a6de06001cf489f34e9

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
ring_buffer.adb:8:21: info: range check proved
ring_buffer.adb:13:61: info: range check proved, in call inlined at ring_buffer.
↳adb:26
ring_buffer.adb:15:16: info: discriminant check proved, in call inlined at ring_
↳buffer.adb:26
ring_buffer.adb:16:45: info: range check proved, in call inlined at ring_buffer.
↳adb:26
ring_buffer.adb:16:45: info: length check proved, in call inlined at ring_buffer.
↳adb:26
ring_buffer.adb:21:25: info: index check proved
ring_buffer.adb:23:37: info: index check proved
ring_buffer.adb:25:24: info: range check proved
ring_buffer.ads:10:07: info: range check proved
ring_buffer.ads:16:13: info: implicit aspect Always_Terminates on "Valid_Model"
↳has been proved, subprogram will terminate
ring_buffer.ads:21:14: info: postcondition proved

```

Everything's fine here. Model is only accessed inside Update_Model, itself a ghost procedure, so it's fine to declare local variable New_Length without the Ghost aspect: everything inside a ghost procedure body is ghost. Moreover, we don't need to add any contract to Update_Model: it's inlined by GNATprove because it's a local procedure without a contract.

63.4.6 Example #6

The function Max_Array takes two arrays of the same length (but not necessarily with the same bounds) as arguments and returns an array with each entry being the maximum values of both arguments at that index.

Listing 1962: array_util.ads

```

1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Max_Array (A, B : Nat_Array) return Nat_Array with
6         Pre => A'Length = B'Length;
7
8 end Array_Util;

```

Listing 1963: array_util.adb

```

1 package body Array_Util is
2
3     function Max_Array (A, B : Nat_Array) return Nat_Array is
4         R : Nat_Array (A'Range);
5         J : Integer := B'First;
6     begin
7         for I in A'Range loop
8             if A (I) > B (J) then
9                 R (I) := A (I);
10            else
11                R (I) := B (J);
12            end if;
13            J := J + 1;
14        end loop;
15        return R;
16    end Max_Array;
17
18 end Array_Util;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_06
MD5: 4b8a6a9b1a3d4d228fe1e944914084fe

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
array_util.adb:8:24: medium: array index check might fail [reason for check: value_
↳ must be a valid index into the array] [possible fix: loop at line 7 should_
↳ mention J in a loop invariant]
array_util.adb:13:17: medium: overflow check might fail, cannot prove upper bound_
↳ for J + 1 [reason for check: result of addition must fit in a 32-bits machine_
↳ integer] [possible fix: loop at line 7 should mention J in a loop invariant]
gnatprove: unproved check messages considered as errors

```

This program is correct, but GNATprove can't prove that J is always in the index range of B (the unproved index check) or even that it's always within the bounds of its type (the unproved overflow check). Indeed, when checking the body of the loop, GNATprove forgets everything about the current value of J because it's been modified by previous loop iterations. To get more precise results, we need to provide a loop invariant.

63.4.7 Example #7

Let's add a loop invariant that states that J stays in the index range of B and let's protect the increment to J by checking that it's not already the maximal integer value.

Listing 1964: array_util.ads

```

1 package Array_Util is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Max_Array (A, B : Nat_Array) return Nat_Array with
6     Pre => A'Length = B'Length;
7
8 end Array_Util;
```

Listing 1965: array_util.adb

```

1 package body Array_Util is
2
3   function Max_Array (A, B : Nat_Array) return Nat_Array is
4     R : Nat_Array (A'Range);
5     J : Integer := B'First;
6   begin
7     for I in A'Range loop
8       pragma Loop_Invariant (J in B'Range);
9       if A (I) > B (J) then
10         R (I) := A (I);
11       else
12         R (I) := B (J);
13       end if;
14       if J < Integer'Last then
15         J := J + 1;
16       end if;
17     end loop;
18     return R;
19   end Max_Array;
20
21 end Array_Util;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_07
MD5: 917629e0683725c23198f8a905a73c57

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
array_util.adb:8:33: medium: loop invariant might not be preserved by an arbitrary_
↪ iteration
gnatprove: unproved check messages considered as errors
```

The loop invariant now allows verifying that no runtime error can occur in the loop's body (property INSIDE seen in section [Loop Invariants](#) (page 1529)). Unfortunately, GNATprove fails to verify that the invariant stays valid after the first iteration of the loop (property PRESERVE). Indeed, knowing that J is in B'Range in a given iteration isn't enough to prove it'll remain so in the next iteration. We need a more precise invariant, linking J to the value of the loop index I, like $J = I - A'First + B'First$.

63.4.8 Example #8

We now consider a version of `Max_Array` which takes arguments that have the same bounds. We want to prove that `Max_Array` returns an array of the maximum values of both its arguments at each index.

Listing 1966: `array_util.ads`

```

1 package Array_Util is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Max_Array (A, B : Nat_Array) return Nat_Array with
6     Pre  => A'First = B'First and A'Last = B'Last,
7     Post => (for all K in A'Range =>
8               Max_Array'Result (K) = Natural'Max (A (K), B (K)));
9
10  end Array_Util;
```

Listing 1967: `array_util.adb`

```

1 package body Array_Util is
2
3   function Max_Array (A, B : Nat_Array) return Nat_Array is
4     R : Nat_Array (A'Range) := (others => 0);
5   begin
6     for I in A'Range loop
7       pragma Loop_Invariant (for all K in A'First .. I =>
8                             R (K) = Natural'Max (A (K), B (K)));
9       if A (I) > B (I) then
10        R (I) := A (I);
11      else
12        R (I) := B (I);
13      end if;
14    end loop;
15    return R;
16  end Max_Array;
17
18 end Array_Util;
```

Listing 1968: `main.adb`

```

1 with Array_Util; use Array_Util;
2
3 procedure Main is
4   A : Nat_Array := (1, 1, 2);
5   B : Nat_Array := (2, 1, 0);
6   R : Nat_Array (1 .. 3);
7 begin
8   R := Max_Array (A, B);
9 end Main;
```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_08
MD5: d0a04c214a632466a4fe4ec6cb7f8842

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

(continues on next page)

(continued from previous page)

```

main.adb:8:09: medium: length check might fail [reason for check: array must be of
↳the appropriate length]
array_util.adb:8:35: medium: loop invariant might fail in first iteration, cannot
↳prove R (K) = Natural'Max (A (K), B (K))
array_util.adb:8:35: medium: loop invariant might not be preserved by an arbitrary
↳iteration, cannot prove R (K) = Natural'Max (A (K), B (K))
gnatprove: unproved check messages considered as errors

```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : Loop_Invariant failed at array_util.adb:7
```

Here, GNATprove doesn't manage to prove the loop invariant even for the first loop iteration (property INIT seen in section [Loop Invariants](#) (page 1529)). In fact, the loop invariant is incorrect, as you can see by executing the function `Max_Array` with assertions enabled: at each loop iteration, `R` contains the maximum of `A` and `B` only until `I - 1` because the `I`'th index wasn't yet handled.

63.4.9 Example #9

We now consider a procedural version of `Max_Array` which updates its first argument instead of returning a new array. We want to prove that `Max_Array` sets the maximum values of both its arguments into each index in its first argument.

Listing 1969: array_util.ads

```

1 package Array_Util is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
6     Pre  => A'First = B'First and A'Last = B'Last,
7     Post => (for all K in A'Range =>
8               A (K) = Natural'Max (A'Old (K), B (K)));
9
10  end Array_Util;

```

Listing 1970: array_util.adb

```

1 package body Array_Util is
2
3   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
4   begin
5     for I in A'Range loop
6       pragma Loop_Invariant
7         (for all K in A'First .. I - 1 =>
8           A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
9       pragma Loop_Invariant
10        (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
11       if A (I) <= B (I) then
12         A (I) := B (I);
13       end if;
14     end loop;
15   end Max_Array;
16
17 end Array_Util;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_09
MD5: 2de4bdd9c59d7d1eccb6259067ffdcf3

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
array_util.adb:7:13: info: loop invariant preservation proved
array_util.adb:7:13: info: loop invariant initialization proved
array_util.adb:7:39: info: overflow check proved
array_util.adb:8:18: info: index check proved
array_util.adb:8:50: info: index check proved
array_util.adb:8:57: info: index check proved
array_util.adb:10:13: info: loop invariant initialization proved
array_util.adb:10:13: info: loop invariant preservation proved
array_util.adb:10:44: info: index check proved
array_util.adb:10:63: info: index check proved
array_util.adb:11:25: info: index check proved
array_util.adb:12:25: info: index check proved
array_util.ads:7:14: info: postcondition proved
array_util.ads:8:20: info: index check proved
array_util.ads:8:45: info: index check proved
array_util.ads:8:52: info: index check proved
```

Everything is proved. The first loop invariant states that the values of A before the loop index contains the maximum values of the arguments of Max_Array (referring to the input value of A with A'Loop_Entry). The second loop invariant states that the values of A beyond and including the loop index are the same as they were on entry. This is the frame condition of the loop.

63.4.10 Example #10

Let's remove the frame condition from the previous example.

Listing 1971: array_util.ads

```
1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
6         Pre  => A'First = B'First and A'Last = B'Last,
7         Post => (for all K in A'Range =>
8                 A (K) = Natural'Max (A'Old (K), B (K)));
9
10    end Array_Util;
```

Listing 1972: array_util.adb

```
1 package body Array_Util is
2
3     procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
4     begin
5         for I in A'Range loop
6             pragma Loop_Invariant
7                 (for all K in A'First .. I - 1 =>
8                 A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
9             if A (I) <= B (I) then
10                A (I) := B (I);
```

(continues on next page)

(continued from previous page)

```

11     end if;
12     end loop;
13     end Max_Array;
14
15 end Array_Util;

```

Code block metadata

Project: Courses.Intro_To_Spark.Proof_of_Functional_Correctness.Example_10
MD5: 8bdc8432cbb3f26f58f63457408c7172

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
array_util.adb:7:13: info: loop invariant preservation proved
array_util.adb:7:13: info: loop invariant initialization proved
array_util.adb:7:39: info: overflow check proved
array_util.adb:8:18: info: index check proved
array_util.adb:8:50: info: index check proved
array_util.adb:8:57: info: index check proved
array_util.adb:9:25: info: index check proved
array_util.adb:10:25: info: index check proved
array_util.ads:7:14: info: postcondition proved
array_util.ads:8:20: info: index check proved
array_util.ads:8:45: info: index check proved
array_util.ads:8:52: info: index check proved

```

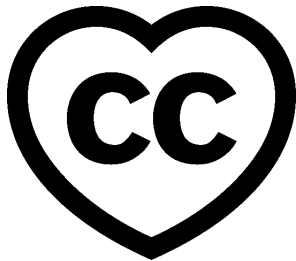
Everything is still proved. GNATprove internally generates the frame condition for the loop, so it's sufficient here to state that A before the loop index contains the maximum values of the arguments of Max_Array.

Part VI

SPARK Ada for the MISRA C Developer

Copyright © 2018 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)⁴¹²



This book presents the SPARK technology — the SPARK subset of Ada and its supporting static analysis tools — through an example-driven comparison with the rules in the widely known MISRA C subset of the C language.

This document was prepared by Yannick Moy, with contributions and review from Ben Brosgol.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](#)⁴¹³. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

⁴¹² [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

⁴¹³ https://learn.adacore.com/zip/learning-ada_code.zip

PREFACE

MISRA C appeared in 1998 as a coding standard for C; it focused on avoiding error-prone programming features of the C programming language rather than on enforcing a particular programming style. A study of coding standards for C by [Les Hatton](https://www.leshatton.org/Documents/MISRAC.pdf)⁴¹⁴ found that, compared to ten typical coding standards for C, MISRA C was the only one to focus exclusively on error avoidance rather than style enforcement, and by a very large margin.

The popularity of the C programming language, as well as its many traps and pitfalls, have led to the huge success of MISRA C in domains where C is used for high-integrity software. This success has driven tool vendors to propose many competing implementations of [MISRA C](https://en.wikipedia.org/wiki/MISRA_C)⁴¹⁵ checkers. Tools compete in particular on the coverage of MISRA C guidelines that they help enforce, as it is impossible to enforce the 16 directives and 143 rules (collectively referred to as guidelines) of MISRA C.

The 16 directives are broad guidelines, and it is not possible to define compliance in a unique and automated way. For example, *"all code should be traceable to documented requirements"* (Directive 3.1). Thus no tool is expected to enforce directives, as the MISRA C:2012 states in introduction to the guidelines: *"different tools may place widely different interpretations on what constitutes a non-compliance."*

The 143 rules on the contrary are completely and precisely defined, and *"static analysis tools should be capable of checking compliance with rules"*. But the same sentence continues with *"subject to the limitations described in Section 6.5"*, which addresses "decidability of rules". It turns out that 27 rules out of 143 are not decidable, so no tool can always detect all violations of these rules without at the same time reporting "false alarms" on code that does not constitute a violation.

An example of an undecidable rule is rule 1.3: *"There shall be no occurrence of undefined or critical unspecified behaviour."* Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable. For the most part, MISRA C checkers ignore undecidable rules such as rule 1.3 and instead focus on the 116 rules for which detection of violations can be automated. It is telling in that respect that the MISRA C:2012 document and its accompanying set of examples (which can be downloaded from the [MISRA website](https://www.misra.org.uk)⁴¹⁶) does not provide any example for rule 1.3.

However, violations of undecidable rules such as rule 1.3 are known to have dramatic impact on software quality. Violations of rule 1.3 in particular are commonly amplified by compilers using the permission in the C standard to optimize aggressively without looking at the consequences for programs with undefined or critical unspecified behavior. It would be valid to ignore these rules if violations did not occur in practice, but on the contrary even experienced programmers write C code with undefined or critical unspecified behavior. An example comes from the MISRA C Committee itself in its "Appendix I: Example deviation record" of the MISRA C:2012 document, repeated in "Appendix A: Example deviation record" of the [MISRA C: Compliance 2016 document](https://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7xA%3d&tabid=57)⁴¹⁷, where the following code is proposed as

⁴¹⁴ <https://www.leshatton.org/Documents/MISRAC.pdf>

⁴¹⁵ https://en.wikipedia.org/wiki/MISRA_C

⁴¹⁶ <https://www.misra.org.uk>

⁴¹⁷ https://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7xA%3d&tabid=57

a deviation of rule 10.6 "*The value of a composite expression shall not be assigned to an object with wider essential type*":

```
uint32_t prod = qty * time_step;
```

Here, the multiplication of two unsigned 16-bit values and assignment of the result to an unsigned 32-bit variable constitutes a violation of the aforementioned rule, which gets justified for efficiency reasons. What the authors seem to have missed is that the multiplication is then performed with the signed integer type `int` instead of the target unsigned type `uint32_t`. Thus the multiplication of two unsigned 16-bit values may lead to an overflow of the 32-bit intermediate signed result, which is an occurrence of an undefined behavior. In such a case, a compiler is free to assume that the value of `prod` cannot exceed $2^{31} - 1$ (the maximal value of a signed 32-bit integer) as otherwise an undefined behavior would have been triggered. For example, the undefined behavior with values 65535 for `qty` and `time_step` is reported when running the code compiled by either the GCC or LLVM compiler with option `-fsanitize=undefined`.

The MISRA C checkers that detect violations of undecidable rules are either unsound tools that can detect only some of the violations, or sound tools that guarantee to detect all such violations at the cost of possibly many false reports of violations. This is a direct consequence of undecidability. However, static analysis technology is available that can achieve soundness without inundating users with false alarms. One example is the SPARK toolset developed by AdaCore, Altran and Inria, which is based on four principles:

- The base language Ada provides a solid foundation for static analysis through a well-defined language standard, strong typing and rich specification features.
- The SPARK subset of Ada restricts the base language in essential ways to support static analysis, by controlling sources of ambiguity such as side-effects and aliasing.
- The static analysis tools work mostly at the granularity of an individual function, making the analysis more precise and minimizing the possibility of false alarms.
- The static analysis tools are interactive, allowing users to guide the analysis if necessary or desired.

In this document, we show how SPARK can be used to achieve high code quality with guarantees that go beyond what would be feasible with MISRA C.

An on-line and interactive version of this document is available at [AdaCore's learn.adacore.com site](https://learn.adacore.com/site)⁴¹⁸.

⁴¹⁸ https://learn.adacore.com/courses/SPARK_for_the_MISRA_C_Developer

ENFORCING BASIC PROGRAM CONSISTENCY

Many consistency properties that are taken for granted in other languages are not enforced in C. The basic property that all uses of a variable or function are consistent with its type is not enforced by the language and is also very difficult to enforce by a tool. Three features of C contribute to that situation:

- the textual-based inclusion of files means that every included declaration is subject to a possibly different reinterpretation depending on context.
- the lack of consistency requirements across translation units means that type inconsistencies can only be detected at link time, something linkers are ill-equipped to do.
- the default of making a declaration externally visible means that declarations that should be local will be visible to the rest of the program, increasing the chances for inconsistencies.

MISRA C contains guidelines on all three fronts to enforce basic program consistency.

65.1 Taming Text-Based Inclusion

The text-based inclusion of files is one of the dated idiosyncracies of the C programming language that was inherited by C++ and that is known to cause quality problems, especially during maintenance. Although multiple inclusion of a file in the same translation unit can be used to emulate template programming, it is generally undesirable. Indeed, MISRA C defines Directive 4.10 precisely to forbid it for header files: *"Precautions shall be taken in order to prevent the contents of a header file being included more than once"*.

The subsequent section on "Preprocessing Directives" contains 14 rules restricting the use of text-based inclusion through preprocessing. Among other things these rules forbid the use of the `#undef` directive (which works around conflicts in macro definitions introduced by text-based inclusion) and enforces the well-known practice of enclosing macro arguments in parentheses (to avoid syntactic reinterpretations in the context of the macro use).

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context instead of textual inclusion of content, using `with` clauses:

Listing 1973: hello_world.adb

```
1 with Ada.Text_IO;  
2  
3 procedure Hello_World is  
4 begin  
5   Ada.Text_IO.Put_Line ("hello, world!");  
6 end Hello_World;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 5ed9609dd61bbcee252bb8529a6d3479

Runtime output

```
hello, world!
```

Note that **with** clauses are only allowed at the beginning of files; the compiler issues an error if they are used elsewhere:

Listing 1974: hello_world.adb

```
1 procedure Hello_World is
2   with Ada.Text_IO; -- Illegal
3 begin
4   Ada.Text_IO.Put_Line ("hello, world!");
5 end Hello_World;
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: afa19e8e2c114a5832b49e9efcbe675e
```

Importing a unit (i.e., specifying it in a **with** clause) multiple times is harmless, as it is equivalent to importing it once, but a compiler warning lets us know about the redundancy:

Listing 1975: hello_world.adb

```
1 with Ada.Text_IO;
2 with Ada.Text_IO; -- Legal but useless
3
4 procedure Hello_World is
5 begin
6   Ada.Text_IO.Put_Line ("hello, world!");
7 end Hello_World;
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 270928968d7beb4809af9e62df530722
```

Runtime output

```
hello, world!
```

The order in which units are imported is irrelevant. All orders are valid and have the same semantics.

No conflict arises from importing multiple units, even if the same name is defined in several, since each unit serves as namespace for the entities which it defines. So we can define our own version of `Put_Line` in some `Helper` unit and import it together with the standard version defined in `Ada.Text_IO`:

Listing 1976: helper.ads

```
1 package Helper is
2   procedure Put_Line (S : String);
3 end Helper;
```

Listing 1977: helper.adb

```
1 with Ada.Text_IO;
2
3 package body Helper is
4   procedure Put_Line (S : String) is
```

(continues on next page)

(continued from previous page)

```

5   begin
6       Ada.Text_IO.Put_Line ("Start helper version");
7       Ada.Text_IO.Put_Line (S);
8       Ada.Text_IO.Put_Line ("End helper version");
9   end Put_Line;
10 end Helper;

```

Listing 1978: hello_world.adb

```

1  with Ada.Text_IO;
2  with Helper;
3
4  procedure Hello_World is
5  begin
6      Ada.Text_IO.Put_Line ("hello, world!");
7      Helper.Put_Line ("hello, world!");
8  end Hello_World;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 5fa012cc996e24e3b1f604e35bbba44f

Runtime output

```

hello, world!
Start helper version
hello, world!
End helper version

```

The only way a conflict can arise is if we want to be able to reference `Put_Line` directly, without using the qualified name `Ada.Text_IO.Put_Line` or `Helper.Put_Line`. The **use** clause makes public declarations from a unit available directly:

Listing 1979: helper.ads

```

1  package Helper is
2      procedure Put_Line (S : String);
3  end Helper;

```

Listing 1980: helper.adb

```

1  with Ada.Text_IO;
2
3  package body Helper is
4      procedure Put_Line (S : String) is
5      begin
6          Ada.Text_IO.Put_Line ("Start helper version");
7          Ada.Text_IO.Put_Line (S);
8          Ada.Text_IO.Put_Line ("End helper version");
9      end Put_Line;
10 end Helper;

```

Listing 1981: hello_world.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Helper; use Helper;
3
4  procedure Hello_World is
5  begin

```

(continues on next page)

(continued from previous page)

```
6 Ada.Text_IO.Put_Line ("hello, world!");
7 Helper.Put_Line ("hello, world!");
8 Put_Line ("hello, world!"); -- ERROR
9 end Hello_World;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 405e138d78e0dc869e8a340681d87e61

Build output

```
hello_world.adb:8:04: error: ambiguous expression (cannot resolve "Put_Line")
hello_world.adb:8:04: error: possible interpretation at helper.ads:2
hello_world.adb:8:04: error: possible interpretation at a-textio.ads:565
gprbuild: *** compilation phase failed
```

Here, both units `Ada.Text_IO` and `Helper` define a procedure `Put_Line` taking a **String** as argument, so the compiler cannot disambiguate the direct call to `Put_Line` and issues an error.

Note that it helpfully points to candidate declarations, so that the user can decide which qualified name to use as in the previous two calls.

Issues arising in C as a result of text-based inclusion of files are thus completely prevented in SPARK (and Ada) thanks to semantic import of units. Note that the C++ committee identified this weakness some time ago and [has approved](#)⁴¹⁹ the addition of *modules* to C++20, which provide a mechanism for semantic import of units.

65.2 Hardening Link-Time Checking

An issue related to text-based inclusion of files is that there is no single source for declaring the type of a variable or function. If a file `origin.c` defines a variable `var` and functions `fun` and `print`:

Listing 1982: `origin.c`

```
1 #include <stdio.h>
2
3 int var = 0;
4 int fun() {
5     return 1;
6 }
7 void print() {
8     printf("var = %d\n", var);
9 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Origin
MD5: 3395f1e43408d5bc5c1e6b8431c959d6

and the corresponding header file `origin.h` declares `var`, `fun` and `print` as having external linkage:

⁴¹⁹ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf>

Listing 1983: origin.h

```

1 extern int var;
2 extern int fun();
3 extern void print();

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Origin
MD5: e8e880a16f5099dc1e0a75ffeeeb9468

then client code can include origin.h with declarations for var and fun:

Listing 1984: main.c

```

1 #include "origin.h"
2
3 int main() {
4     var = fun();
5     print();
6     return 0;
7 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Origin
MD5: 3d4582d3956897b657778ae355d0ef1b

Runtime output

```
var = 1
```

or, equivalently, repeat these declarations directly:

Listing 1985: main.c

```

1 extern int var;
2 extern int fun();
3 extern void print();
4
5 int main() {
6     var = fun();
7     print();
8     return 0;
9 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Origin
MD5: 4b25aa011b580f92f2831a48008fbef6

Runtime output

```
var = 1
```

Then, if an inconsistency is introduced in the type of var or fun between these alternative declarations and their actual type, the compiler cannot detect it. Only the linker, which has access to the set of object files for a program, can detect such inconsistencies. However, a linker's main task is to link, not to detect inconsistencies, and so inconsistencies in the type of variables and functions in most cases cannot be detected. For example, most linkers cannot detect if the type of var or the return type of fun is changed to **float** in the declarations above. With the declaration of var changed to **float**, the above program compiles

and runs without errors, producing the erroneous output `var = 1065353216` instead of `var = 1`. With the return type of `fun` changed to `float` instead, the program still compiles and runs without errors, producing this time the erroneous output `var = 0`.

The inconsistency just discussed is prevented by MISRA C Rule 8.3 "All declarations of an object or function shall use the same names and type qualifiers". This is a decidable rule, but it must be enforced at system level, looking at all translation units of the complete program. MISRA C Rule 8.6 also requires a unique definition for a given identifier across translation units, and Rule 8.5 requires that an external declaration shared between translation units comes from the same file. There is even a specific section on "Identifiers" containing 9 rules requiring uniqueness of various categories of identifiers.

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context using `with` clauses to provide a unique declaration for each entity.

65.3 Going Towards Encapsulation

Many problems in C stem from the lack of encapsulation. There is no notion of namespace that would allow a file to make its declarations available without risking a conflict with other files. Thus MISRA C has a number of guidelines that discourage the use of external declarations:

- Directive 4.8 encourages hiding the definition of structures and unions in implementation files (`.c` files) when possible: *"If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden."*
- Rule 8.7 forbids the use of external declarations when not needed: *"Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."*
- Rule 8.8 forces the explicit use of keyword `static` when appropriate: *"The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage."*

The basic unit of modularization in SPARK, as in Ada, is the *package*. A package always has a spec (in an `.ads` file), which defines the interface to other units. It generally also has a body (in an `.adb` file), which completes the spec with an implementation. Only declarations from the package spec are visible from other units when they import (`with`) the package. In fact, only declarations from what is called the "visible part" of the spec (before the keyword `private`) are visible from units that `with` the package.

Listing 1986: helper.ads

```
1 package Helper is
2   procedure Public_Put_Line (S : String);
3 private
4   procedure Private_Put_Line (S : String);
5 end Helper;
```

Listing 1987: helper.adb

```
1 with Ada.Text_IO;
2
3 package body Helper is
4   procedure Public_Put_Line (S : String) is
5   begin
6     Ada.Text_IO.Put_Line (S);
7   end Public_Put_Line;
```

(continues on next page)

(continued from previous page)

```

8
9  procedure Private_Put_Line (S : String) is
10 begin
11     Ada.Text_IO.Put_Line (S);
12 end Private_Put_Line;
13
14 procedure Body_Put_Line (S : String) is
15 begin
16     Ada.Text_IO.Put_Line (S);
17 end Body_Put_Line;
18 end Helper;

```

Listing 1988: hello_world.adb

```

1  with Helper; use Helper;
2
3  procedure Hello_World is
4  begin
5      Public_Put_Line ("hello, world!");
6      Private_Put_Line ("hello, world!"); -- ERROR
7      Body_Put_Line ("hello, world!"); -- ERROR
8  end Hello_World;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hello_World
MD5: 148fd8101cc72413909675534f5e359c

Build output

```

hello_world.adb:6:04: error: "Private_Put_Line" is not visible
hello_world.adb:6:04: error: non-visible (private) declaration at helper.ads:4
hello_world.adb:7:04: error: "Body_Put_Line" is undefined
gprbuild: *** compilation phase failed

```

Note the different errors on the calls to the private and body versions of `Put_Line`. In the first case the compiler can locate the candidate procedure but it is illegal to call it from `Hello_World`, in the second case the compiler does not even know about any `Body_Put_Line` when compiling `Hello_World` since it only looks at the spec and not the body.

SPARK (and Ada) also allow defining a type in the private part of a package spec while simply declaring the type name in the public ("visible") part of the spec. This way, client code — i.e., code that `with`'s the package — can use the type, typically through a public API, but have no access to how the type is implemented:

Listing 1989: vault.ads

```

1  package Vault is
2      type Data is private;
3      function Get (X : Data) return Integer;
4      procedure Set (X : out Data; Value : Integer);
5  private
6      type Data is record
7          Val : Integer;
8      end record;
9  end Vault;

```

Listing 1990: vault.adb

```
1 package body Vault is
2   function Get (X : Data) return Integer is (X.Val);
3   procedure Set (X : out Data; Value : Integer) is
4   begin
5     X.Val := Value;
6   end Set;
7 end Vault;
```

Listing 1991: information_system.ads

```
1 with Vault;
2
3 package Information_System is
4   Archive : Vault.Data;
5 end Information_System;
```

Listing 1992: hacker.adb

```
1 with Information_System;
2 with Vault;
3
4 procedure Hacker is
5   V : Integer := Vault.Get (Information_System.Archive);
6 begin
7   Vault.Set (Information_System.Archive, V + 1);
8   Information_System.Archive.Val := 0; -- ERROR
9 end Hacker;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Program_Consistency.Hacker
MD5: 065ed34dc727e2eb0bdc50a667cb1f78

Build output

```
hacker.adb:8:22: error: invalid prefix in selected component "Information_System.
↪Archive"
gprbuild: *** compilation phase failed
```

Note that it is possible to declare a variable of type `Vault.Data` in package `Information_System` and to get/set it through its API in procedure `Hacker`, but not to directly access its `Val` field.

ENFORCING BASIC SYNTACTIC GUARANTEES

C's syntax is concise but also very permissive, which makes it easy to write programs whose effect is not what was intended. MISRA C contains guidelines to:

- clearly distinguish code from comments
- specially handle function parameters and result
- ensure that control structures are not abused

66.1 Distinguishing Code and Comments

The problem arises from block comments in C, starting with `/*` and ending with `*/`. These comments do not nest with other block comments or with line comments. For example, consider a block comment surrounding three lines that each increase variable `a` by one:

```
/*  
++a;  
++a;  
++a; */
```

Now consider what happens if the first line is commented out using a block comment and the third line is commented out using a line comment (also known as a C++ style comment, allowed in C since C99):

```
/*  
/* ++a; */  
++a;  
// ++a; */
```

The result of commenting out code that was already commented out is that the second line of code becomes live! Of course, the above example is simplified, but similar situations do arise in practice, which is the reason for MISRA C Directive 4.1 "*Sections of code should not be 'commented out'*". This is reinforced with Rules 3.1 and 3.2 from the section on "Comments" that forbid in particular the use of `/*` inside a comment like we did above.

These situations cannot arise in SPARK (or in Ada), as only line comments are permitted, using `--`:

```
-- A := A + 1;  
-- A := A + 1;  
-- A := A + 1;
```

So commenting again the first and third lines does not change the effect:

```
-- -- A := A + 1;  
-- A := A + 1;  
-- -- A := A + 1;
```

66.2 Specially Handling Function Parameters and Result

66.2.1 Handling the Result of Function Calls

It is possible in C to ignore the result of a function call, either implicitly or else explicitly by converting the result to **void**:

```
f();
(void)f();
```

This is particularly dangerous when the function returns an error status, as the caller is then ignoring the possibility of errors in the callee. Thus the MISRA C Directive 4.7: *"If a function returns error information, then that error information shall be tested"*. In the general case of a function returning a result which is not an error status, MISRA C Rule 17.7 states that *"The value returned by a function having non-void return type shall be used"*, where an explicit conversion to **void** counts as a use.

In SPARK, as in Ada, the result of a function call must be used, for example by assigning it to a variable or by passing it as a parameter, in contrast with procedures (which are equivalent to void-returning functions in C). SPARK analysis also checks that the result of the function is actually used to influence an output of the calling subprogram. For example, the first two calls to F in the following are detected as unused, even though the result of the function call is assigned to a variable, which is itself used in the second case:

Listing 1993: fun.ads

```
1 package Fun is
2   function F return Integer is (1);
3 end Fun;
```

Listing 1994: use_f.ads

```
1 procedure Use_F (Z : out Integer);
```

Listing 1995: use_f.adb

```
1 with Fun; use Fun;
2
3 procedure Use_F (Z : out Integer) is
4   X, Y : Integer;
5 begin
6   X := F;
7
8   Y := F;
9   X := Y;
10
11  Z := F;
12 end Use_F;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Func_Return
MD5: 4fc78b4136677d6338984ab8ccfa5cd1

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
use_f.adb:6:06: warning: unused assignment
```

(continues on next page)

(continued from previous page)

```
use_f.adb:8:06: warning: unused assignment
use_f.adb:9:06: warning: unused assignment
```

Only the result of the third call is used to influence the value of an output of Use_F, here the output parameter Z of the procedure.

66.2.2 Handling Function Parameters

In C, function parameters are treated as local variables of the function. They can be modified, but these modifications won't be visible outside the function. This is an opportunity for mistakes. For example, the following code, which appears to swap the values of its parameters, has in reality no effect:

```
void swap (int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

MISRA C Rule 17.8 prevents such mistakes by stating that "A function parameter should not be modified".

No such rule is needed in SPARK, since function parameters are only inputs so cannot be modified, and procedure parameters have a *mode* defining whether they can be modified or not. Only parameters of mode **out** or *ada:in out* can be modified — and these are prohibited from functions in SPARK — and their modification is visible at the calling site. For example, assigning to a parameter of mode **in** (the default parameter mode if omitted) results in compilation errors:

Listing 1996: swap.ads

```
1 procedure Swap (X, Y : Integer);
```

Listing 1997: swap.adb

```
1 procedure Swap (X, Y : Integer) is
2   Tmp : Integer := X;
3 begin
4   X := Y; -- ERROR
5   Y := Tmp; -- ERROR
6 end Swap;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Swap
MD5: 187927c610e202f2e1eee6a602fda25e

Build output

```
swap.adb:4:04: error: assignment to "in" mode parameter not allowed
swap.adb:5:04: error: assignment to "in" mode parameter not allowed
gprbuild: *** compilation phase failed
```

Here is the output of AdaCore's GNAT compiler:

```
1. procedure Swap (X, Y : Integer) is
2.   Tmp : Integer := X;
3.   begin
4.     X := Y; -- ERROR
```

(continues on next page)

(continued from previous page)

```

      |
>>> assignment to "in" mode parameter not allowed
5.      Y := Tmp;  -- ERROR
      |
>>> assignment to "in" mode parameter not allowed
6.      end Swap;

```

The correct version of Swap in SPARK takes parameters of mode **in out**:

Listing 1998: swap.ads

```
1 procedure Swap (X, Y : in out Integer);
```

Listing 1999: swap.adb

```

1 procedure Swap (X, Y : in out Integer) is
2   Tmp : constant Integer := X;
3 begin
4   X := Y;
5   Y := Tmp;
6 end Swap;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Swap
MD5: c983a229fc5a69db5dbb85f49a91b325

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...

66.3 Ensuring Control Structures Are Not Abused

The previous issue (ignoring the result of a function call) is an example of a control structure being abused, due to the permissive syntax of C. There are many such examples, and MISRA C contains a number of guidelines to prevent such abuse.

66.3.1 Preventing the Semicolon Mistake

Because a semicolon can act as a statement, and because an if-statement and a loop accept a simple statement (possibly only a semicolon) as body, inserting a single semicolon can completely change the behavior of the code:

```

int func() {
  if (0)
    return 1;
  while (1)
    return 0;
  return 0;
}

```

As written, the code above returns with status 0. If a semicolon is added after the first line (**if** (0);), then the code returns with status 1. If a semicolon is added instead after the third line (**while** (1);), then the code does not return. To prevent such surprises, MISRA C

Rule 15.6 states that "*The body of an iteration-statement or a selection-statement shall be a compound statement*" so that the code above must be written:

```
int func() {
    if (0) {
        return 1;
    }
    while (1) {
        return 0;
    }
    return 0;
}
```

Note that adding a semicolon after the test of the **if** or **while** statement has the same effect as before! But doing so would violate MISRA C Rule 15.6.

In SPARK, the semicolon is not a statement by itself, but rather a marker that terminates a statement. The null statement is an explicit **null**;, and all blocks of statements have explicit **begin** and **end** markers, which prevents mistakes that are possible in C. The SPARK (also Ada) version of the above C code is as follows:

Listing 2000: func.ads

```
1 function Func return Integer;
```

Listing 2001: func.adb

```
1 function Func return Integer is
2 begin
3     if False then
4         return 1;
5     end if;
6     while True loop
7         return 0;
8     end loop;
9     return 0;
10 end Func;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Semicolon
MD5: 34fc5967c41d337aada17429ee5f44e9

Prover output

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
func.adb:3:04: warning: statement has no effect
func.adb:4:07: warning: this statement is never reached

66.3.2 Avoiding Complex Switch Statements

Switch statements are well-known for being easily misused. Control can jump to any case section in the body of the switch, which in C can be before any statement contained in the body of the switch. At the end of the sequence of statements associated with a case, execution continues with the code that follows unless a **break** is encountered. This is a recipe for mistakes, and MISRA C enforces a simpler *well-formed* syntax for switch statements defined in Rule 16.1: "*All switch statements shall be well-formed*".

The other rules in the section on "Switch statements" go on detailing individual consequences of Rule 16.1. For example Rule 16.3 forbids the fall-through from one case to the

next: "An unconditional break statement shall terminate every switch-clause". As another example, Rule 16.4 mandates the presence of a default case to handle cases not taken into account explicitly: "Every switch statement shall have a default label".

The analog of the C switch statements in SPARK (and in Ada) is the case statement. This statement has a simpler and more robust structure than the C switch, with control automatically exiting after one of the case alternatives is executed, and the compiler checking that the alternatives are disjoint (like in C) and complete (unlike in C). So the following code is rejected by the compiler:

Listing 2002: sign_domain.ads

```
1 package Sign_Domain is
2
3   type Sign is (Negative, Zero, Positive);
4
5   function Opposite (A : Sign) return Sign is
6     (case A is -- ERROR
7       when Negative => Positive,
8       when Positive => Negative);
9
10  function Multiply (A, B : Sign) return Sign is
11    (case A is
12     when Negative      => Opposite (B),
13     when Zero | Positive => Zero,
14     when Positive      => B); -- ERROR
15
16  procedure Get_Sign (X : Integer; S : out Sign);
17
18 end Sign_Domain;
```

Listing 2003: sign_domain.adb

```
1 package body Sign_Domain is
2
3   procedure Get_Sign (X : Integer; S : out Sign) is
4   begin
5     case X is
6       when 0 => S := Zero;
7       when others => S := Negative; -- ERROR
8       when 1 .. Integer'Last => S := Positive;
9     end case;
10  end Get_Sign;
11
12 end Sign_Domain;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Case_Statement
MD5: d345a4d23b5b2402f8bd103e5e550a3b

Build output

```
sign_domain.adb:7:15: error: the choice "others" must appear alone and last
sign_domain.ads:6:07: error: missing case value: "Zero"
sign_domain.ads:14:15: error: duplication of choice value: "Positive" at line 13
gprbuild: *** compilation phase failed
```

The error in function Opposite is that the **when** choices do not cover all values of the target expression. Here, A is of the enumeration type Sign, so all three values of the enumeration must be covered.

The error in function Multiply is that **Positive** is covered twice, in the second and the

third alternatives. This is not allowed.

The error in procedure `Get_Sign` is that the **others** choice (the equivalent of C **default** case) must come last. Note that an **others** choice would be useless in `Opposite` and `Multiply`, as all `Sign` values are covered.

Here is a correct version of the same code:

Listing 2004: `sign_domain.ads`

```

1 package Sign_Domain is
2
3     type Sign is (Negative, Zero, Positive);
4
5     function Opposite (A : Sign) return Sign is
6     (case A is
7         when Negative => Positive,
8         when Zero      => Zero,
9         when Positive  => Negative);
10
11    function Multiply (A, B : Sign) return Sign is
12    (case A is
13        when Negative => Opposite (B),
14        when Zero      => Zero,
15        when Positive  => B);
16
17    procedure Get_Sign (X : Integer; S : out Sign);
18
19 end Sign_Domain;
```

Listing 2005: `sign_domain.adb`

```

1 package body Sign_Domain is
2
3     procedure Get_Sign (X : Integer; S : out Sign) is
4     begin
5         case X is
6             when 0 => S := Zero;
7             when 1 .. Integer'Last => S := Positive;
8             when others => S := Negative;
9         end case;
10    end Get_Sign;
11
12 end Sign_Domain;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Case_Statement
MD5: 1c99fc53d2d2c0dddbea5e5b0a6c5746

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
sign_domain.ads:5:13: info: implicit aspect Always_Terminates on "Opposite" has_
↳ been proved, subprogram will terminate
sign_domain.ads:11:13: info: implicit aspect Always_Terminates on "Multiply" has_
↳ been proved, subprogram will terminate
sign_domain.ads:17:37: info: initialization of "S" proved
```

66.3.3 Avoiding Complex Loops

Similarly to C switches, for-loops in C can become unreadable. MISRA C thus enforces a simpler *well-formed* syntax for for-loops, defined in Rule 14.2: "A for loop shall be well-formed". The main effect of this simplification is that for-loops in C look like for-loops in SPARK (and in Ada), with a *loop counter* that is incremented or decremented at each iteration. Section 8.14 defines precisely what a loop counter is:

1. It has a scalar type;
2. Its value varies monotonically on each loop iteration; and
3. It is used in a decision to exit the loop.

In particular, Rule 14.2 forbids any modification of the loop counter inside the loop body. Here's the example used in MISRA C:2012 to illustrate this rule:

```
bool_t flag = false;

for ( int16_t i = 0; ( i < 5 ) && !flag; i++ )
{
    if ( C )
    {
        flag = true; /* Compliant - allows early termination of loop */
    }

    i = i + 3;      /* Non-compliant - altering the loop counter */
}
```

The equivalent SPARK (and Ada) code does not compile, because of the attempt to modify the value of the loop counter:

Listing 2006: well_formed_loop.adb

```
1  procedure Well_Formed_Loop (C : Boolean) is
2      Flag : Boolean := False;
3  begin
4      for I in 0 .. 4 loop
5          exit when not Flag;
6
7          if C then
8              Flag := True;
9          end if;
10
11         I := I + 3;  -- ERROR
12     end loop;
13 end Well_Formed_Loop;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Well_Formed_Loop
MD5: 842564c961aa018e03e03f81439995ec

Build output

```
well_formed_loop.adb:11:07: error: assignment to loop parameter not allowed
gprbuild: *** compilation phase failed
```

Removing the problematic line leads to a valid program. Note that the additional condition being tested in the C for-loop has been moved to a separate exit statement at the start of the loop body.

SPARK (and Ada) loops can increase (or, with explicit syntax, decrease) the loop counter by 1 at each iteration.

```

for I in reverse 0 .. 4 loop
    ... -- Successive values of I are 4, 3, 2, 1, 0
end loop;

```

SPARK loops can iterate over any discrete type; i.e., integers as above or enumerations:

```

type Sign is (Negative, Zero, Positive);

for S in Sign loop
    ...
end loop;

```

66.3.4 Avoiding the Dangling Else Issue

C does not provide a closing symbol for an if-statement. This makes it possible to write the following code, which appears to try to return the absolute value of its argument, while it actually does the opposite:

Listing 2007: main.c

```

1  #include <stdio.h>
2
3  int absval (int x) {
4      int result = x;
5      if (x >= 0)
6          if (x == 0)
7              result = 0;
8      else
9          result = -x;
10     return result;
11 }
12
13 int main() {
14     printf("absval(5) = %d\n", absval(5));
15     printf("absval(0) = %d\n", absval(0));
16     printf("absval(-10) = %d\n", absval(-10));
17 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Dangling_Else_C
MD5: c180a948dd8bed4e3b97efde1522214c

Runtime output

```

absval(5) = -5
absval(0) = 0
absval(-10) = -10

```

The warning issued by GCC or LLVM with option `-Wdangling-else` (implied by `-Wall`) gives a clue about the problem: although the `else` branch is written as though it completes the outer if-statement, in fact it completes the inner if-statement.

MISRA C Rule 15.6 avoids the problem: *"The body of an iteration-statement or a selection-statement shall be a compound statement"*. That's the same rule as the one shown earlier for [Preventing the Semicolon Mistake](#) (page 1566). So the code for `absval` must be written:

Listing 2008: main.c

```

1  #include <stdio.h>
2

```

(continues on next page)

(continued from previous page)

```

3  int absval (int x) {
4      int result = x;
5      if (x >= 0) {
6          if (x == 0) {
7              result = 0;
8          }
9      } else {
10         result = -x;
11     }
12     return result;
13 }
14
15 int main() {
16     printf("absval(5) = %d\n", absval(5));
17     printf("absval(0) = %d\n", absval(0));
18     printf("absval(-10) = %d\n", absval(-10));
19 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Dangling_Else_ MISRA_C
MD5: 2b76377aca52ff45ed6b19fa1f367473

Runtime output

```

absval(5) = 5
absval(0) = 0
absval(-10) = 10

```

which has the expected behavior.

In SPARK (as in Ada), each if-statement has a matching end marker **end if**; so the dangling-else problem cannot arise. The above C code is written as follows:

Listing 2009: absval.ads

```

1  function Absval (X : Integer) return Integer;

```

Listing 2010: absval.adb

```

1  function Absval (X : Integer) return Integer is
2      Result : Integer := X;
3  begin
4      if X >= 0 then
5          if X = 0 then
6              Result := 0;
7          end if;
8      else
9          Result := -X;
10     end if;
11     return Result;
12 end Absval;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Syntactic_Guarantees.Dangling_Else_Ada
MD5: e867b6354ef7bdd89bae1673e888153a

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
absval.adb:9:17: medium: overflow check might fail, cannot prove upper bound for -
↳X [reason for check: result of negation must fit in a 32-bits machine integer]↳
↳[possible fix: add precondition (-X <= Integer'Last) to subprogram at absval.
↳ads:1]
gnatprove: unproved check messages considered as errors
```

Interestingly, SPARK analysis detects here that the negation operation on line 9 might overflow. That's an example of runtime error detection which will be covered in the chapter on [*Detecting Undefined Behavior*](#) (page 1609).

ENFORCING STRONG TYPING

Annex C of MISRA C:2012 summarizes the problem succinctly:

"ISO C may be considered to exhibit poor type safety as it permits a wide range of implicit type conversions to take place. These type conversions can compromise safety as their implementation-defined aspects can cause developer confusion."

The most severe consequences come from inappropriate conversions involving pointer types, as they can cause memory safety violations. Two sections of MISRA C are dedicated to these issues: "Pointer type conversions" (9 rules) and "Pointers and arrays" (8 rules).

Inappropriate conversions between scalar types are only slightly less severe, as they may introduce arbitrary violations of the intended functionality. MISRA C has gone to great lengths to improve the situation, by defining a stricter type system on top of the C language. This is described in Appendix D of MISRA C:2012 and in the dedicated section on "The essential type model" (8 rules).

67.1 Enforcing Strong Typing for Pointers

Pointers in C provide a low-level view of the addressable memory as a set of integer addresses. To write at address 42, just go through a pointer:

Listing 2011: main.c

```
1 int main() {  
2     int *p = 42;  
3     *p = 0;  
4     return 0;  
5 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Pointers_C
MD5: 005183ada50cb6642f38a3640d77efff

Running this program is likely to hit a segmentation fault on an operating system, or to cause havoc in an embedded system, both because address 42 will not be correctly aligned on a 32-bit or 64-bit machine and because this address is unlikely to correspond to valid addressable data for the application. The compiler might issue a helpful warning on the above code (with option `-Wint-conversion` implied by `-Wall` in GCC or LLVM), but note that the warning disappears when explicitly converting value 42 to the target pointer type, although the problem is still present.

Beyond their ability to denote memory addresses, pointers are also used in C to pass references as inputs or outputs to function calls, to construct complex data structures with indirection or sharing, and to denote arrays of elements. Pointers are thus at once pervasive, powerful and fragile.

67.1.1 Pointers Are Not Addresses

In an attempt to rule out issues that come from direct addressing of memory with pointers, MISRA C states in Rule 11.4 that *"A conversion should not be performed between a pointer to object and an integer type"*. As this rule is classified as only Advisory, MISRA C completes it with two Required rules:

- Rule 11.6: *"A cast shall not be performed between pointer to void and an arithmetic type"*
- Rule 11.7: *"A cast shall not be performed between pointer to object and a non-integer arithmetic type"*

In Ada, pointers are not addresses, and addresses are not integers. An opaque standard type `System.Address` is used for addresses, and conversions to/from integers are provided in a standard package `System.Storage_Elements`. The previous C code can be written as follows in Ada:

Listing 2012: pointer.adb

```

1 with System;
2 with System.Storage_Elements;
3
4 procedure Pointer is
5   A : constant System.Address := System.Storage_Elements.To_Address (42);
6   M : aliased Integer with Address => A;
7   P : constant access Integer := M'Access;
8 begin
9   P.all := 0;
10 end Pointer;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Pointers_Ada
MD5: 32ac91ade61a39d3505d155d7b97a8a5

The integer value 42 is converted to a memory address A by calling `System.Storage_Elements.To_Address`, which is then used as the address of integer variable M. The pointer variable P is set to point to M (which is allowed because M is declared as **aliased**).

Ada requires more verbiage than C:

- The integer value 42 must be explicitly converted to type **Address**
- To get a pointer to a declared variable such as M, the declaration must be marked as **aliased**

The added syntax helps first in making clear what is happening and, second, in ensuring that a potentially dangerous feature (assigning to a value at a specific machine address) is not used inadvertently.

The above example is legal in SPARK, but the SPARK analysis tool issues warnings as it cannot control how the program or its environment may update the memory cell at address 42.

67.1.2 Pointers Are Not References

Passing parameters by reference is critical for efficient programs, but the absence of references distinct from pointers in C incurs a serious risk. Any parameter of a pointer type can be copied freely to a variable whose lifetime is longer than the object pointed to, a problem known as "dangling pointers". MISRA C forbids such uses in Rule 18.6: *"The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist"*. Unfortunately, enforcing this rule is difficult, as it is undecidable.

In SPARK, parameters can be passed by reference, but no pointer to the parameter can be stored past the return point of the function, which completely solves this issue. In fact, the decision to pass a parameter by copy or by reference rests in many cases with the compiler, but such compiler dependency has no effect on the functional behavior of a SPARK program. In the example below, the compiler may decide to pass parameter P of procedure Rotate_X either by copy or by reference, but regardless of the choice the postcondition of Rotate_X will hold: the final value of P will be modified by rotation around the X axis.

Listing 2013: geometry.ads

```

1 package Geometry is
2
3   type Point_3D is record
4     X, Y, Z : Float;
5   end record;
6
7   procedure Rotate_X (P : in out Point_3D) with
8     Post => P = P'Old'Update (Y => P.Z'Old, Z => -P.Y'Old);
9
10 end Geometry;
```

Listing 2014: geometry.adb

```

1 package body Geometry is
2
3   procedure Rotate_X (P : in out Point_3D) is
4     Tmp : constant Float := P.Y;
5   begin
6     P.Y := P.Z;
7     P.Z := -Tmp;
8   end Rotate_X;
9
10 end Geometry;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Geometry
MD5: d3801cf1413887ffd5fff8b6b86b7742

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
geometry.ads:8:14: info: postcondition proved

SPARK's analysis tool can mathematically prove that the postcondition is true.

67.1.3 Pointers Are Not Arrays

The greatest source of vulnerabilities regarding pointers is their use as substitutes for arrays. Although the C language has a syntax for declaring and accessing arrays, this is just a thin syntactic layer on top of pointers. Thus:

- Array access is just pointer arithmetic;
- If a function is to manipulate an array then the array's length must be separately passed as a parameter; and
- The program is susceptible to the various vulnerabilities originating from the confusion of pointers and arrays, such as buffer overflow.

Consider a function that counts the number of times a value is present in an array. In C, this could be written:

Listing 2015: main.c

```
1  #include <stdio.h>
2
3  int count(int *p, int len, int v) {
4      int count = 0;
5      while (len--) {
6          if (*p++ == v) {
7              count++;
8          }
9      }
10     return count;
11 }
12
13 int main() {
14     int p[5] = {0, 3, 9, 3, 3};
15     int c = count(p, 5, 3);
16     printf("value 3 is seen %d times in p\n", c);
17     return 0;
18 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_C
MD5: 34e3f7c2352e89a7c834184761293e57

Runtime output

value 3 is seen 3 times in p

Function `count` has no control over the range of addresses accessed from pointer `p`. The critical property that the `len` parameter is a valid length for an array of integers pointed to by parameter `p` rests completely with the caller of `count`, and `count` has no way to check that this is true.

To mitigate the risks associated with pointers being used for arrays, MISRA C contains eight rules in a section on "Pointers and arrays". These rules forbid pointer arithmetic (Rule 18.4) or, if this Advisory rule is not followed, require pointer arithmetic to stay within bounds (Rule 18.1). But, even if we rewrite the loop in `count` to respect all decidable MISRA C rules, the program's correctness still depends on the caller of `count` passing a correct value of `len`:

Listing 2016: main.c

```
1  #include <stdio.h>
2
3  int count(int *p, int len, int v) {
4      int count = 0;
5      for (int i = 0; i < len; i++) {
6          if (p[i] == v) {
7              count++;
8          }
9      }
10     return count;
11 }
12
13 int main() {
14     int p[5] = {0, 3, 9, 3, 3};
15     int c = count(p, 5, 3);
16     printf("value 3 is seen %d times in p\n", c);
```

(continues on next page)

(continued from previous page)

```

17   return 0;
18 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_MISRA_C
MD5: d04179de3f1e309541b3d88e53eb5e3a

Runtime output

```
value 3 is seen 3 times in p
```

The resulting code is more readable, but still vulnerable to incorrect values of parameter `len` passed by the caller of `count`, which violates undecidable MISRA C Rules 18.1 (pointer arithmetic should stay within bounds) and 1.3 (no undefined behavior). Contrast this with the same function in SPARK (and Ada):

Listing 2017: types.ads

```

1 package Types is
2   type Int_Array is array (Positive range <>) of Integer;
3 end Types;

```

Listing 2018: count.ads

```

1 with Types; use Types;
2
3 function Count (P : Int_Array; V : Integer) return Natural;

```

Listing 2019: count.adb

```

1 function Count (P : Int_Array; V : Integer) return Natural is
2   Count : Natural := 0;
3 begin
4   for I in P'Range loop
5     if P (I) = V then
6       Count := Count + 1;
7     end if;
8   end loop;
9   return Count;
10 end Count;

```

Listing 2020: test_count.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Types; use Types;
3 with Count;
4
5 procedure Test_Count is
6   P : constant Int_Array := (0, 3, 9, 3, 3);
7   C : constant Integer := Count (P, 3);
8 begin
9   Put_Line ("value 3 is seen" & C'Img & " times in p");
10 end Test_Count;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_Ada
MD5: 82e9d18d4b8ad8aa87ca8520bd7b830c

Runtime output

```
value 3 is seen 3 times in p
```

The array parameter `P` is not simply a homogeneous sequence of Integer values. The compiler must represent `P` so that its lower and upper bounds (`P'First` and `P'Last`) and thus also its length (`P'Length`) can be retrieved. Function `Count` can simply loop over the range of valid array indexes `P'First .. P'Last` (or `P'Range` for short). As a result, function `Count` can be verified in isolation to be free of vulnerabilities such as buffer overflow, as it does not depend on the values of parameters passed by its callers. In fact, we can go further in SPARK and show that the value returned by `Count` is no greater than the length of parameter `P` by stating this property in the postcondition of `Count` and asking the SPARK analysis tool to prove it:

Listing 2021: types.ads

```
1 package Types is
2   type Int_Array is array (Positive range <>) of Integer;
3 end Types;
```

Listing 2022: count.ads

```
1 with Types; use Types;
2
3 function Count (P : Int_Array; V : Integer) return Natural with
4   Post => Count'Result <= P'Length;
```

Listing 2023: count.adb

```
1 function Count (P : Int_Array; V : Integer) return Natural
2 is
3   Count : Natural := 0;
4 begin
5   for I in P'Range loop
6     pragma Loop_Invariant (Count <= I - P'First);
7     if P (I) = V then
8       Count := Count + 1;
9     end if;
10  end loop;
11  return Count;
12 end Count;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Arrays_Ada
MD5: 4c9a34614d53c4d268cbff787c9b73e6

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
count.adb:6:30: info: loop invariant initialization proved
count.adb:6:30: info: loop invariant preservation proved
count.adb:6:41: info: overflow check proved
count.adb:8:25: info: overflow check proved
count.ads:3:10: info: implicit aspect Always_Terminates on "Count" has been proved,
↳ subprogram will terminate
count.ads:4:11: info: postcondition proved
count.ads:4:28: info: range check proved
test_count.adb:9:32: info: range check proved
```

(continues on next page)

(continued from previous page)

test_count.adb:9:40: info: range check proved

The only help that SPARK analysis required from the programmer, in order to prove the postcondition, is a loop invariant (a special kind of assertion) that reflects the value of **Count** at each iteration.

67.1.4 Pointers Should Be Typed

The C language defines a special pointer type **void*** that corresponds to an untyped pointer. It is legal to convert any pointer type to and from **void***, which makes it a convenient way to simulate C++ style templates. Consider the following code which indirectly applies `assign_int` to integer `i` and `assign_float` to floating-point `f` by calling `assign` on both:

Listing 2024: main.c

```

1  #include <stdio.h>
2
3  void assign_int (int *p) {
4      *p = 42;
5  }
6
7  void assign_float (float *p) {
8      *p = 42.0;
9  }
10
11 typedef void (*assign_fun)(void *p);
12
13 void assign(assign_fun fun, void *p) {
14     fun(p);
15 }
16
17 int main() {
18     int i;
19     float f;
20     assign((assign_fun)&assign_int, &i);
21     assign((assign_fun)&assign_float, &f);
22     printf("i = %d; f = %f\n", i, f);
23 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Typed_Pointers_C
MD5: fc00ba9eb97640037488569347591cc2

Runtime output

```
i = 42; f = 42.000000
```

The references to the variables `i` and `f` are implicitly converted to the **void*** type as a way to apply `assign` to any second parameter `p` whose type matches the argument type of its first argument `fun`. The use of an untyped argument means that the responsibility for the correct typing rests completely with the programmer. Swap `i` and `f` in the calls to `assign` and you still get a compilable program without warnings, that runs and produces completely bogus output:

```
i = 1109917696; f = 0.000000
```

instead of the expected:


```
i = 42; f = 42.000000
```

Generics in SPARK (and Ada) can implement the desired functionality in a fully typed way, with any errors caught at compile time, where procedure `Assign` applies its parameter procedure `Initialize` to its parameter `V`:

Listing 2025: assign.ads

```
1 generic
2   type T is private;
3   with procedure Initialize (V : out T);
4   procedure Assign (V : out T);
```

Listing 2026: assign.adb

```
1 procedure Assign (V : out T) is
2   begin
3     Initialize (V);
4   end Assign;
```

Listing 2027: apply_assign.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Assign;
3
4 procedure Apply_Assign is
5   procedure Assign_Int (V : out Integer) is
6     begin
7       V := 42;
8     end Assign_Int;
9
10    procedure Assign_Float (V : out Float) is
11      begin
12        V := 42.0;
13      end Assign_Float;
14
15    procedure Assign_I is new Assign (Integer, Assign_Int);
16    procedure Assign_F is new Assign (Float, Assign_Float);
17
18    I : Integer;
19    F : Float;
20  begin
21    Assign_I (I);
22    Assign_F (F);
23    Put_Line ("I =" & I'Img & "; F =" & F'Img);
24  end Apply_Assign;
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Typed_Pointers_Ada
MD5: af23d6f8a742676139aac38a385c7bf7
```

Runtime output

```
I = 42; F = 4.20000E+01
```

The generic procedure `Assign` must be instantiated with a specific type for `T` and a specific procedure (taking a single `out` parameter of this type) for `Initialize`. The procedure resulting from the instantiation applies to a variable of this type. So switching `I` and `F` above would result in an error detected by the compiler. Likewise, an instantiation such as the following would also be a compile-time error:

```
procedure Assign_I is new Assign (Integer, Assign_Float);
```

67.2 Enforcing Strong Typing for Scalars

In C, all scalar types can be converted both implicitly and explicitly to any other scalar type. The semantics is defined by rules of *promotion* and *conversion*, which can confuse even experts. One example was noted earlier, in the *Preface* (page 1553). Another example appears in *an article introducing a safe library for manipulating scalars*⁴²⁰ by Microsoft expert David LeBlanc. In its conclusion, the author acknowledges the inherent difficulty in understanding scalar type conversions in C, by showing an early buggy version of the code to produce the minimum signed integer:

```
return (T)(1 << (BitCount()-1));
```

The issue here is that the literal `1` on the left-hand side of the shift is an `int`, so on a 64-bit machine with 32-bit `int` and 64-bit type `T`, the above is shifting 32-bit value `1` by 63 bits. This is a case of undefined behavior, producing an unexpected output with the Microsoft compiler. The correction is to convert the first literal `1` to `T` before the shift:

```
return (T)((T)1 << (BitCount()-1));
```

Although he'd asked some expert programmers to review the code, no one found this problem.

To avoid these issues as much as possible, MISRA C defines its own type system on top of C types, in the section on "The essential type model" (eight rules). These can be seen as additional typing rules, since all rules in this section are decidable, and can be enforced at the level of a single translation unit. These rules forbid in particular the confusing cases mentioned above. They can be divided into three sets of rules:

- restricting operations on types
- restricting explicit conversions
- restricting implicit conversions

67.2.1 Restricting Operations on Types

Apart from the application of some operations to floating-point arguments (the bitwise, mod and array access operations) which are invalid and reported by the compiler, all operations apply to all scalar types in C. MISRA C Rule 10.1 constrains the types on which each operation is possible as follows.

Arithmetic Operations on Arithmetic Types

Adding two Boolean values, or an Apple and an Orange, might sound like a bad idea, but it is easily done in C:

Listing 2028: main.c

```
1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
5     bool b1 = true;
6     bool b2 = false;
```

(continues on next page)

⁴²⁰ <https://msdn.microsoft.com/en-us/library/ms972705.aspx>

(continued from previous page)

```
7  bool b3 = b1 + b2;
8
9  typedef enum {Apple, Orange} fruit;
10 fruit f1 = Apple;
11 fruit f2 = Orange;
12 fruit f3 = f1 + f2;
13
14 printf("b3 = %d; f3 = %d\n", b3, f3);
15
16 return 0;
17 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Pointer_Arith_C
MD5: 30e28b34f616f8e6d35233a4ce698c23

Runtime output

```
b3 = 1; f3 = 1
```

No error from the compiler here. In fact, there is no undefined behavior in the above code. Variables `b3` and `f3` both end up with value 1. Of course it makes no sense to add Boolean or enumerated values, and thus MISRA C Rule 18.1 forbids the use of all arithmetic operations on Boolean and enumerated values, while also forbidding most arithmetic operations on characters. That leaves the use of arithmetic operations for signed or unsigned integers as well as floating-point types and the use of modulo operation `%` for signed or unsigned integers.

Here's an attempt to simulate the above C code in SPARK (and Ada):

Listing 2029: bad_arith.ads

```
1 package Bad_Arith is
2
3   B1 : constant Boolean := True;
4   B2 : constant Boolean := False;
5   B3 : constant Boolean := B1 + B2;
6
7   type Fruit is (Apple, Orange);
8   F1 : constant Fruit := Apple;
9   F2 : constant Fruit := Orange;
10  F3 : constant Fruit := F1 + F2;
11
12 end Bad_Arith;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Pointer_Arith_Ada
MD5: 984381fdcf1a682e1998f7881c0532f9

Build output

```
bad_arith.ads:5:32: error: there is no applicable operator "+" for type "Standard.
↳ Boolean"
bad_arith.ads:10:30: error: there is no applicable operator "+" for type "Fruit"
↳ defined at line 7
gprbuild: *** compilation phase failed
```

It is possible, however, to get the predecessor of a Boolean or enumerated value with Value '`Pred`' and its successor with Value '`Succ`', as well as to iterate over all values of the

type:

Listing 2030: ok_arith.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Ok_Arith is
4
5     B1 : constant Boolean := False;
6     B2 : constant Boolean := Boolean'Succ (B1);
7     B3 : constant Boolean := Boolean'Pred (B2);
8
9     type Fruit is (Apple, Orange);
10    F1 : constant Fruit := Apple;
11    F2 : constant Fruit := Fruit'Succ (F1);
12    F3 : constant Fruit := Fruit'Pred (F2);
13
14 begin
15     pragma Assert (B1 = B3);
16     pragma Assert (F1 = F3);
17
18     for B in Boolean loop
19         Put_Line (B'Img);
20     end loop;
21
22     for F in Fruit loop
23         Put_Line (F'Img);
24     end loop;
25 end Ok_Arith;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Pointer_Arith_Ada
MD5: 6ad400913a48fd815845b6a99d90ec2d

Runtime output

```

FALSE
TRUE
APPLE
ORANGE
```

Boolean Operations on Boolean

"Two bee or not two bee? Let's C":

Listing 2031: main.c

```

1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
5     typedef enum {Ape, Bee, Cat} Animal;
6     bool answer = (2 * Bee) || ! (2 * Bee);
7     printf("two bee or not two bee? %d\n", answer);
8     return 0;
9 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Boolean_C
MD5: a9d4886827c983df51c9285fe3fd6c77

Runtime output

```
two bee or not two bee? 1
```

The answer to the question posed by Shakespeare's Hamlet is 1, since it reduces to A **or not** A and this is true in classical logic.

As previously noted, MISRA C forbids the use of the multiplication operator with an operand of an enumerated type. Rule 18.1 also forbids the use of Boolean operations "and", "or", and "not" (&&, ||, !, respectively, in C) on anything other than Boolean operands. It would thus prohibit the Shakespearian code above.

Below is an attempt to express the same code in SPARK (and Ada), where the Boolean operators are **and**, **or**, and **not**. The **and** and **or** operators evaluate both operands, and the language also supplies short-circuit forms that evaluate the left operand and only evaluate the right operand when its value may affect the result.

Listing 2032: bad_hamlet.ads

```
1 package Bad_Hamlet is
2   type Animal is (Ape, Bee, Cat);
3   Answer : Boolean := 2 * Bee or not 2 * Bee; -- Illegal
4 end Bad_Hamlet;
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Boolean_Ada
MD5: 9089114f9cc6495dabd6957b54b33bd2
```

Build output

```
bad_hamlet.ads:3:28: error: expected type universal integer
bad_hamlet.ads:3:28: error: found type "Animal" defined at line 2
bad_hamlet.ads:3:43: error: expected a modular type
bad_hamlet.ads:3:43: error: found type "Animal" defined at line 2
gprbuild: *** compilation phase failed
```

As expected, the compiler rejects this code. There is no available `*` operation that works on an enumeration type, and likewise no available **or** or **not** operation.

Bitwise Operations on Unsigned Integers

Here's a genetic engineering example that combines a Bee with a Dog to produce a Cat, by manipulating the atomic structure (the bits in its representation):

Listing 2033: main.c

```
1 #include <stdbool.h>
2 #include <assert.h>
3
4 int main() {
5     typedef enum {Ape, Bee, Cat, Dog} Animal;
6     Animal mutant = Bee ^ Dog;
7     assert (mutant == Cat);
8     return 0;
9 }
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Bitwise_C
MD5: 645b0b6155f1cb17d02c7bcbb976993c
```

This algorithm works by accessing the underlying bitwise representation of Bee and Dog (0x01 and 0x03, respectively) and, by applying the exclusive-or operator `^`, transforming it into the underlying bitwise representation of a Cat (0x02). While powerful, manipulating the bits in the representation of values is best reserved for unsigned integers as illustrated in the book [Hacker's Delight](http://www.hackersdelight.org/)⁴²¹. MISRA C Rule 18.1 thus forbids the use of all bitwise operations on anything but unsigned integers.

Below is an attempt to do the same in SPARK (and Ada). The bitwise operators are `and`, `or`, `xor`, and `not`, and the related bitwise functions are `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`:

Listing 2034: bad_genetics.ads

```

1 package Bad_Genetics is
2   type Animal is (Ape, Bee, Cat, Dog);
3   Mutant : Animal := Bee xor Dog; -- ERROR
4   pragma Assert (Mutant = Cat);
5 end Bad_Genetics;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Bitwise_Ada
MD5: 3f7c3dd616f065016590d574200cf1db

Build output

```

bad_genetics.ads:3:27: error: there is no applicable operator "Xor" for type
↳ "Animal" defined at line 2
gprbuild: *** compilation phase failed
```

The declaration of Mutant is illegal, since the `xor` operator is only available for Boolean and unsigned integer (modular) values; it is not available for `Animal`. The same restriction applies to the other bitwise operators listed above. If we really wanted to achieve the effect of the above code in legal SPARK (or Ada), then the following approach will work (the type `Unsigned_8` is an 8-bit modular type declared in the predefined package `Interfaces`).

Listing 2035: unethical_genetics.ads

```

1 with Interfaces; use Interfaces;
2 package Unethical_Genetics is
3   type Animal is (Ape, Bee, Cat, Dog);
4   A      : constant array (Animal) of Unsigned_8 :=
5           (Animal'Pos (Ape), Animal'Pos (Bee),
6            Animal'Pos (Cat), Animal'Pos (Dog));
7   Mutant : Animal := Animal'Val (A (Bee) xor A (Dog));
8   pragma Assert (Mutant = Cat);
9 end Unethical_Genetics;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Bitwise_Ada_2
MD5: 359439d40740fe2d99e6f334ed3500f9

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
```

Note that `and`, `or`, `not` and `xor` are used both as logical operators and as bitwise operators, but there is no possible confusion between these two uses. Indeed the use of such operators

⁴²¹ <http://www.hackersdelight.org/>

on values from modular types is a natural generalization of their uses on Boolean, since values from modular types are often interpreted as arrays of Booleans.

67.2.2 Restricting Explicit Conversions

A simple way to bypass the restrictions of Rule 10.1 is to explicitly convert the arguments of an operation to a type that the rule allows. While it can often be useful to cast a value from one type to another, many casts that are allowed in C are either downright errors or poor replacements for clearer syntax.

One example is to cast from a scalar type to Boolean. A better way to express `(bool)x` is to compare `x` to the zero value of its type: `x != 0` for integers, `x != 0.0` for floats, `x != '0'` for characters, `x != Enum` where `Enum` is the first enumerated value of the type. Thus, MISRA C Rule 10.5 advises avoiding casting non-Boolean values to Boolean.

Rule 10.5 also advises avoiding other casts that are, at best, obscure:

- from a Boolean to any other scalar type
- from a floating-point value to an enumeration or a character
- from any scalar type to an enumeration

The rules are not symmetric, so although a float should not be cast to an enum, casting an enum to a float is allowed. Similarly, although it is advised to not cast a character to an enum, casting an enum to a character is allowed.

The rules in SPARK are simpler. There are no conversions between numeric types (integers, fixed-point and floating-point) and non-numeric types (such as Boolean, Character, and other enumeration types). Conversions between different non-numeric types are limited to those that make semantic sense, for example between a derived type and its parent type. Any numeric type can be converted to any other numeric type, with precise rules for rounding/truncating values when needed and run-time checking that the converted value is in the range associated with the target type.

67.2.3 Restricting Implicit Conversions

Rules 10.1 and 10.5 restrict operations on types and explicit conversions. That's not enough to avoid problematic C programs; a program violating one of these rules can be expressed using only implicit type conversions. For example, the Shakespearian code in section [Boolean Operations on Boolean](#) (page 1585) can be reformulated to satisfy both Rules 10.1 and 10.5:

Listing 2036: main.c

```
1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
5     typedef enum {Ape, Bee, Cat} Animal;
6     int b = Bee;
7     bool t = 2 * b;
8     bool answer = t || ! t;
9     printf("two bee or not two bee? %d\n", answer);
10    return 0;
11 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Implicit_Conversion_C
MD5: a157dd05c5fe8926886361b533305e14

Runtime output

```
two bee or not two bee? 1
```

Here, we're implicitly converting the enumerated value `Bee` to an `int`, and then implicitly converting the integer value `2 * b` to a `Boolean`. This does not violate 10.1 or 10.5, but it is prohibited by MISRA C Rule 10.3: *"The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category"*.

Rule 10.1 also does not prevent arguments of an operation from being inconsistent, for example comparing a floating-point value and an enumerated value. But MISRA C Rule 10.4 handles this situation: *"Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category"*.

In addition, three rules in the "Composite operators and expressions" section avoid common mistakes related to the combination of explicit/implicit conversions and operations.

The rules in SPARK (and Ada) are far simpler: there are no implicit conversions! This applies both between types of a different *essential type category* as MISRA C puts it, as well as between types that are structurally the same but declared as different types.

Listing 2037: `bad_conversions.adb`

```

1  procedure Bad_Conversions is
2      pragma Warnings (Off);
3      F : Float := 0.0;
4      I : Integer := 0;
5      type Animal is (Ape, Bee, Cat);
6      type My_Animal is new Animal; -- derived type
7      A : Animal := Cat;
8      M : My_Animal := Bee;
9      B : Boolean := True;
10     C : Character := 'a';
11 begin
12     F := I; -- ERROR
13     I := A; -- ERROR
14     A := B; -- ERROR
15     M := A; -- ERROR
16     B := C; -- ERROR
17     C := F; -- ERROR
18 end Bad_Conversions;
```

Code block metadata

Project: `Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Implicit_Conversion_Bad_Ada`
MD5: `f10b50048595df0b4ed77c06a7508412`

Build output

```

bad_conversions.adb:12:09: error: expected type "Standard.Float"
bad_conversions.adb:12:09: error: found type "Standard.Integer"
bad_conversions.adb:13:09: error: expected type "Standard.Integer"
bad_conversions.adb:13:09: error: found type "Animal" defined at line 5
bad_conversions.adb:14:09: error: expected type "Animal" defined at line 5
bad_conversions.adb:14:09: error: found type "Standard.Boolean"
bad_conversions.adb:15:09: error: expected type "My_Animal" defined at line 6
bad_conversions.adb:15:09: error: found type "Animal" defined at line 5
bad_conversions.adb:16:09: error: expected type "Standard.Boolean"
bad_conversions.adb:16:09: error: found type "Standard.Character"
bad_conversions.adb:17:09: error: expected type "Standard.Character"
bad_conversions.adb:17:09: error: found type "Standard.Float"
gprbuild: *** compilation phase failed
```


The compiler reports a mismatch on every statement in the above procedure (the declarations are all legal).

Adding explicit conversions makes the assignments to F and M valid, since SPARK (and Ada) allow conversions between numeric types and between a derived type and its parent type, but all other conversions are illegal:

Listing 2038: bad_conversions.adb

```
1 procedure Bad_Conversions is
2   pragma Warnings (Off);
3   F : Float := 0.0;
4   I : Integer := 0;
5   type Animal is (Ape, Bee, Cat);
6   type My_Animal is new Animal; -- derived type
7   A : Animal := Cat;
8   M : My_Animal := Bee;
9   B : Boolean := True;
10  C : Character := 'a';
11 begin
12  F := Float (I);           -- OK
13  I := Integer (A);         -- ERROR
14  A := Animal (B);          -- ERROR
15  M := My_Animal (A);       -- OK
16  B := Boolean (C);         -- ERROR
17  C := Character (F);       -- ERROR
18 end Bad_Conversions;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Strong_Typing.Implicit_Conversion_Bad_Ada
MD5: 4d3f6a8629d51f27b6628dae5fc7b680

Build output

```
bad_conversions.adb:13:18: error: illegal operand for numeric conversion
bad_conversions.adb:14:09: error: invalid conversion, not compatible with type
↳ "Standard.Boolean"
bad_conversions.adb:16:09: error: invalid conversion, not compatible with type
↳ "Standard.Character"
bad_conversions.adb:17:09: error: invalid conversion, not compatible with type
↳ "Standard.Float"
gprbuild: *** compilation phase failed
```

Although an enumeration value cannot be converted to an integer (or *vice versa*) either implicitly or explicitly, SPARK (and Ada) provide functions to obtain the effect of a type conversion. For any enumeration type T, the function T'Pos(e) takes an enumeration value from type T and returns its relative position as an integer, starting at 0. For example, Animal'Pos(Bee) is 1, and Boolean'Pos(False) is 0. In the other direction, T'Val(n), where n is an integer, returns the enumeration value in type T at relative position n. If n is negative or greater than T'Pos(T'Last) then a run-time exception is raised.

Hence, the following is valid SPARK (and Ada) code; Character is defined as an enumeration type:

Listing 2039: ok_conversions.adb

```
1 procedure Ok_Conversions is
2   pragma Warnings (Off);
3   F : Float := 0.0;
4   I : Integer := 0;
```

(continues on next page)

(continued from previous page)

```
5  type Animal is (Ape, Bee, Cat);
6  type My_Animal is new Animal;
7  A : Animal := Cat;
8  M : My_Animal := Bee;
9  B : Boolean := True;
10 C : Character := 'a';
11 begin
12   F := Float (I);
13   I := Animal'Pos (A);
14   I := My_Animal'Pos (M);
15   I := Boolean'Pos (B);
16   I := Character'Pos (C);
17   I := Integer (F);
18   A := Animal'Val (2);
19 end Ok_Conversions;
```


INITIALIZING DATA BEFORE USE

As with most programming languages, C does not require that variables be initialized at their declaration, which makes it possible to unintentionally read uninitialized data. This is a case of undefined behavior, which can sometimes be used to attack the program.

68.1 Detecting Reads of Uninitialized Data

MISRA C attempts to prevent reads of uninitialized data in a specific section on "Initialization", containing five rules. The most important is Rule 9.1: *"The value of an object with automatic storage duration shall not be read before it has been set"*. The first example in the rule is interesting, as it shows a non-trivial (and common) case of conditional initialization, where a function `f` initializes an output parameter `p` only in some cases, and the caller `g` of `f` ends up reading the value of the variable `u` passed in argument to `f` in cases where it has not been initialized:

Listing 2040: `f.h`

```
1 #include <stdint.h>
2
3 void f ( int b, uint16_t *p );
```

Listing 2041: `f.c`

```
1 #include "f.h"
2
3 void f ( int b, uint16_t *p )
4 {
5     if ( b )
6     {
7         *p = 3U;
8     }
9 }
```

Listing 2042: `g.c`

```
1 #include <stdint.h>
2 #include "f.h"
3
4 static void g (void)
5 {
6     uint16_t u;
7
8     f ( 0, &u );
9
10    if ( u == 3U )
11    {
```

(continues on next page)

(continued from previous page)

```
12      /* Non-compliant use - "u" has not been assigned a value. */
13  }
14 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_C
MD5: f36430141f48b34810d53a43294c7d74

Detecting the violation of Rule 9.1 can be arbitrarily complex, as the program points corresponding to a variable's initialization and read can be separated by many calls and conditions. This is one of the undecidable rules, for which most MISRA C checkers won't detect all violations.

In SPARK, the guarantee that all reads are to initialized data is enforced by the SPARK analysis tool, GNATprove, through what is referred to as *flow analysis*. Every subprogram is analyzed separately to check that it cannot read uninitialized data. To make this modular analysis possible, SPARK programs need to respect the following constraints:

- all inputs of a subprogram should be initialized on subprogram entry
- all outputs of a subprogram should be initialized on subprogram return

Hence, given the following code translated from C, GNATprove reports that function F might not always initialize output parameter P:

Listing 2043: init.ads

```
1 with Interfaces; use Interfaces;
2
3 package Init is
4   procedure F (B : Boolean; P : out Unsigned_16);
5   procedure G;
6 end Init;
```

Listing 2044: init.adb

```
1 package body Init is
2
3   procedure F (B : Boolean; P : out Unsigned_16) is
4   begin
5     if B then
6       P := 3;
7     end if;
8   end F;
9
10  procedure G is
11    U : Unsigned_16;
12  begin
13    F (False, U);
14
15    if U = 3 then
16      null;
17    end if;
18  end G;
19
20 end Init;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_↵Ada
 MD5: d54bc9901b3bffa4f0cfea9942a795156

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 init.ads:4:30: medium: "P" might not be initialized in "F" [reason for check: OUT_↵
 ↵parameter should be initialized on return] [possible fix: initialize "P" on all_↵
 ↵paths or make "P" an IN OUT parameter]
 gnatprove: unproved check messages considered as errors

We can correct the program by initializing P to value 0 when condition B is not satisfied:

Listing 2045: init.ads

```
1 with Interfaces; use Interfaces;
2
3 package Init is
4   procedure F (B : Boolean; P : out Unsigned_16);
5   procedure G;
6 end Init;
```

Listing 2046: init.adb

```
1 package body Init is
2
3   procedure F (B : Boolean; P : out Unsigned_16) is
4   begin
5     if B then
6       P := 3;
7     else
8       P := 0;
9     end if;
10  end F;
11
12  procedure G is
13    U : Unsigned_16;
14  begin
15    F (False, U);
16
17    if U = 3 then
18      null;
19    end if;
20  end G;
21
22 end Init;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_↵Ada
 MD5: 481787c333014d56814a7205720f72bc

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...

(continues on next page)

(continued from previous page)

```
init.adb:13:07: info: initialization of "U" proved  
init.ads:4:30: info: initialization of "P" proved
```

GNATprove now does not report any possible reads of uninitialized data. On the contrary, it confirms that all reads are made from initialized data.

In contrast with C, SPARK does not guarantee that global data (called *library-level* data in SPARK and Ada) is zero-initialized at program startup. Instead, GNATprove checks that all global data is explicitly initialized (at declaration or elsewhere) before it is read. Hence it goes beyond the MISRA C Rule 9.1, which considers global data as always initialized even if the default value of all-zeros might not be valid data for the application. Here's a variation of the above code where variable U is now global:

Listing 2047: init.ads

```
1 with Interfaces; use Interfaces;  
2  
3 package Init is  
4   U : Unsigned_16;  
5   procedure F (B : Boolean);  
6   procedure G;  
7 end Init;
```

Listing 2048: init.adb

```
1 package body Init is  
2  
3   procedure F (B : Boolean) is  
4   begin  
5     if B then  
6       U := 3;  
7     end if;  
8   end F;  
9  
10  procedure G is  
11  begin  
12    F (False);  
13  
14    if U = 3 then  
15      null;  
16    end if;  
17  end G;  
18  
19 end Init;
```

Listing 2049: call_init.adb

```
1 with Init;  
2  
3 procedure Call_Init is  
4 begin  
5   Init.G;  
6 end Call_Init;
```

Code block metadata

```
Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_  
↳Ada  
MD5: a85cde45a658727975367b041a1a5dc3
```

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
init.adb:14:07: warning: statement has no effect
call_init.adb:5:08: medium: "U" might not be initialized after elaboration of main_
↳ program "Call_Init"
gnatprove: unproved check messages considered as errors

```

GNATprove reports here that variable U might not be initialized at program startup, which is indeed the case here. It reports this issue on the main program `Call_Init` because its analysis showed that F needs to take U as an initialized input (since F is not initializing U on all paths, U keeps its value on the other path, which needs to be an initialized value), which means that G which calls F also needs to take U as an initialized input, which in turn means that `Call_Init` which calls G also needs to take U as an initialized input. At this point, we've reached the main program, so the initialization phase (referred to as *elaboration* in SPARK and Ada) should have taken care of initializing U. This is not the case here, hence the message from GNATprove.

It is possible in SPARK to specify that G should initialize variable U; this is done with a *data dependency* contract introduced with aspect `Global` following the declaration of procedure G:

Listing 2050: init.ads

```

1 with Interfaces; use Interfaces;
2
3 package Init is
4     U : Unsigned_16;
5     procedure F (B : Boolean);
6     procedure G with Global => (Output => U);
7 end Init;

```

Listing 2051: init.adb

```

1 package body Init is
2
3     procedure F (B : Boolean) is
4     begin
5         if B then
6             U := 3;
7         end if;
8     end F;
9
10    procedure G is
11    begin
12        F (False);
13
14        if U = 3 then
15            null;
16        end if;
17    end G;
18
19 end Init;

```

Listing 2052: call_init.adb

```

1 with Init;
2
3 procedure Call_Init is
4 begin
5     Init.G;

```

(continues on next page)

(continued from previous page)

6 `end Call_Init;`**Code block metadata**

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Read_Uninitialized_Data_
 ↪Ada
 MD5: 100122ca3c8c60c134822a85d564a60a

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 init.adb:12:07: high: "U" is not an input in the Global contract of subprogram "G" ↪
 ↪at init.ads:6
 init.adb:12:07: high: either make "U" an input in the Global contract or ↪
 ↪initialize it before use
 init.adb:14:07: warning: statement has no effect
 gnatprove: unproved check messages considered as errors

GNATprove reports the error on the call to F in G, as it knows at this point that F needs U to be initialized but the calling context in G cannot provide that guarantee. If we provide the same data dependency contract for F, then GNATprove reports the error on F itself, similarly to what we saw for an output parameter U.

68.2 Detecting Partial or Redundant Initialization of Arrays and Structures

The other rules in the section on "Initialization" deal with common errors in initializing aggregates and *designated initializers* in C99 to initialize a structure or array at declaration. These rules attempt to patch holes created by the lax syntax and rules in C standard. For example, here are five valid initializations of an array of 10 elements in C:

Listing 2053: main.c

```
1 int main() {
2     int a[10] = {0};
3     int b[10] = {0, 0};
4     int c[10] = {0, [8] = 0};
5     int d[10] = {0, [8] = 0, 0};
6     int e[10] = {0, [8] = 0, 0, [8] = 1};
7     return 0;
8 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Redundant_Init
 MD5: 1212a5565fc3a382e7f967d1cf0b48f9

Only a is fully initialized to all-zeros in the above code snippet. MISRA C Rule 9.3 thus forbids all other declarations by stating that "*Arrays shall not be partially initialized*". In addition, MISRA C Rule 9.4 forbids the declaration of e by stating that "*An element of an object shall not be initialised more than once*" (in e's declaration, the element at index 8 is initialized twice).

The same holds for initialization of structures. Here is an equivalent set of declarations with the same potential issues:

Listing 2054: main.c

```

1 int main() {
2     typedef struct { int x; int y; int z; } rec;
3     rec a = {0};
4     rec b = {0, 0};
5     rec c = {0, .y = 0};
6     rec d = {0, .y = 0, 0};
7     rec e = {0, .y = 0, 0, .y = 1};
8     return 0;
9 }

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Redundant_Init
MD5: e562ef70b8c8a170d2bd09281cf2a075

Here only a, d and e are fully initialized. MISRA C Rule 9.3 thus forbids the declarations of b and c. In addition, MISRA C Rule 9.4 forbids the declaration of e.

In SPARK and Ada, the aggregate used to initialize an array or a record must fully cover the components of the array or record. Violations lead to compilation errors, both for records:

Listing 2055: init_record.ads

```

1 package Init_Record is
2     type Rec is record
3         X, Y, Z : Integer;
4     end record;
5     R : Rec := (X => 1); -- ERROR, Y and Z not specified
6 end Init_Record;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_1
MD5: 6b28bffe6270c5ea5055123c5b89c508

Build output

```

init_record.ads:5:15: error: no value supplied for component "Y"
init_record.ads:5:15: error: no value supplied for component "Z"
gprbuild: *** compilation phase failed

```

and for arrays:

Listing 2056: init_array.ads

```

1 package Init_Array is
2     type Arr is array (1 .. 10) of Integer;
3     A : Arr := (1 => 1); -- ERROR, elements 2..10 not specified
4 end Init_Array;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Array_1
MD5: 81aa6363ba770ded10bef8d3d8776914

Build output

```

init_array.ads:3:15: warning: too few elements for type "Arr" defined at line 2
↳[enabled by default]
init_array.ads:3:15: warning: expected 10 elements; found 1 element [enabled by
(continues on next page)

```

(continued from previous page)

```
↪default]
init_array.ads:3:15: warning: Constraint_Error will be raised at run time [enabled_
↪by default]
```

Similarly, redundant initialization leads to compilation errors for records:

Listing 2057: init_record.ads

```
1 package Init_Record is
2   type Rec is record
3     X, Y, Z : Integer;
4   end record;
5   R : Rec := (X => 1, Y => 1, Z => 1, X => 2); -- ERROR, X duplicated
6 end Init_Record;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_2
MD5: 07d3f790009be97cef2daaf08b2f7afd

Build output

```
init_record.ads:5:40: error: more than one value supplied for "X"
gprbuild: *** compilation phase failed
```

and for arrays:

Listing 2058: init_array.ads

```
1 package Init_Array is
2   type Arr is array (1 .. 10) of Integer;
3   A : Arr := (1 .. 8 => 1, 9 .. 10 => 2, 7 => 3); -- ERROR, A(7) duplicated
4 end Init_Array;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Array_2
MD5: 12f5fa4615abccde43f63f72340fd4a0

Build output

```
init_array.ads:3:43: error: index value in array aggregate duplicates the one_
↪given at line 3
init_array.ads:3:43: error: 7
gprbuild: *** compilation phase failed
```

Finally, while it is legal in Ada to leave uninitialized parts in a record or array aggregate by using the box notation (meaning that the default initialization of the type is used, which may be no initialization at all), SPARK analysis rejects such use when it leads to components not being initialized, both for records:

Listing 2059: init_record.ads

```
1 package Init_Record is
2   type Rec is record
3     X, Y, Z : Integer;
4   end record;
5   R : Rec := (X => 1, others => <>); -- ERROR, Y and Z not specified
6 end Init_Record;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Initialization.Init_Record_3
 MD5: a7736f2b563c39fb4ab10007e927ad97

Prover output

```
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
init_record.ads:5:04: error: "R" is not allowed in SPARK (due to box notation,
↳without default or relaxed initialization)
init_record.ads:5:04: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_
↳Record_3/a7736f2b563c39fb4ab10007e927ad97/main_spark.adc:12
init_record.ads:5:15: error: box notation without default or relaxed
↳initialization is not allowed in SPARK (SPARK RM 4.3(1))
init_record.ads:5:15: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_
↳Record_3/a7736f2b563c39fb4ab10007e927ad97/main_spark.adc:12
gnatprove: error during analysis of data and information flow
```

and for arrays:

Listing 2060: init_array.ads

```
1 package Init_Array is
2   type Arr is array (1 .. 10) of Integer;
3   A : Arr := (1 .. 8 => 1, 9 .. 10 => <>); -- ERROR, A(9..10) not specified
4 end Init_Array;
```


CONTROLLING SIDE EFFECTS

As with most programming languages, C allows side effects in expressions. This leads to subtle issues about conflicting side effects, when subexpressions of the same expression read/write the same variable.

69.1 Preventing Undefined Behavior

Conflicting side effects are a kind of undefined behavior; the C Standard (section 6.5) defines the concept as follows:

"Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored"

This legalistic wording is somewhat opaque, but the notion of sequence points is summarized in Annex C of the C90 and C99 standards. MISRA C repeats these conditions in the Amplification of Rule 13.2, including the read of a volatile variable as a side effect similar to writing a variable.

This rule is undecidable, so MISRA C completes it with two rules that provide simpler restrictions preventing some side effects in expressions, thus reducing the potential for undefined behavior:

- Rule 13.3: *"A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator"*.
- Rule 13.4: *"The result of an assignment operator should not be used"*.

In practice, conflicting side effects usually manifest themselves as portability issues, since the result of the evaluation of an expression depends on the order in which a compiler decides to evaluate its subexpressions. So changing the compiler version or the target platform might lead to a different behavior of the application.

To reduce the dependency on evaluation order, MISRA C Rule 13.1 states: *"Initializer lists shall not contain persistent side effects"*. This case is theoretically different from the previously mentioned conflicting side effects, because initializers that comprise an initializer list are separated by sequence points, so there is no risk of undefined behavior if two initializers have conflicting side effects. But given that initializers are executed in an unspecified order, the result of a conflict is potentially as damaging for the application.

69.2 Reducing Programmer Confusion

Even in cases with no undefined or unspecified behavior, expressions with multiple side effects can be confusing to programmers reading or maintaining the code. This problem arises in particular with C's increment and decrement operators that can be applied prior to or after the expression evaluation, and with the assignment operator = in C since it can easily be mistaken for equality. Thus MISRA C forbids the use of the increment / decrement

(Rule 13.3) and assignment (Rule 13.4) operators in expressions that have other potential side effects.

In other cases, the presence of expressions with side effects might be confusing, if the programmer wrongly thinks that the side effects are guaranteed to occur. Consider the function `decrease_until_one_is_null` below, which decreases both arguments until one is null:

Listing 2061: main.c

```
1  #include <stdio.h>
2
3  void decrease_until_one_is_null (int *x, int *y) {
4      if (x == 0 || y == 0) {
5          return;
6      }
7      while (--*x != 0 && --*y != 0) {
8          // nothing
9      }
10 }
11
12 int main() {
13     int x = 42, y = 42;
14     decrease_until_one_is_null (&x, &y);
15     printf("x = %d, y = %d\n", x, y);
16     return 0;
17 }
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Side_Effect_C
MD5: a3e991881894bc3fb25a5f49a083fd2e

Runtime output

x = 0, y = 1

The program produces the following output:

x = 0, y = 1

I.e., starting from the same value 42 for both `x` and `y`, only `x` has reached the value zero after `decrease_until_one_is_null` returns. The reason is that the side effect on `y` is performed only conditionally. To avoid such surprises, MISRA C Rule 13.5 states: "*The right hand operand of a logical && or || operator shall not contain persistent side effects*"; this rule forbids the code above.

MISRA C Rule 13.6 similarly states: "*The operand of the sizeof operator shall not contain any expression which has potential side effects*". Indeed, the operand of `sizeof` is evaluated only in rare situations, and only according to C99 rules, which makes any side effect in such an operand a likely mistake.

69.3 Side Effects and SPARK

In SPARK, expressions cannot have side effects; only statements can. In particular, there are no increment/decrement operators, and no assignment operator. There is instead an assignment statement, whose syntax using `:=` clearly distinguishes it from equality (using `=`). And in any event an expression is not allowed as a statement and this a construct such as `X = Y;` would be illegal. Here is how a variable `X` can be assigned, incremented and decremented:

```

X := 1;
X := X + 1;
X := X - 1;

```

There are two possible side effects when evaluating an expression:

- a read of a volatile variable
- a side effect occurring inside a function that the expression calls

Reads of volatile variables in SPARK are restricted to appear immediately at statement level, so the following is not allowed:

Listing 2062: volatile_read.ads

```

1 package Volatile_Read is
2   X : Integer with Volatile;
3   procedure P (Y : out Integer);
4 end Volatile_Read;

```

Listing 2063: volatile_read.adb

```

1 package body Volatile_Read is
2   procedure P (Y : out Integer) is
3   begin
4     Y := X - X; -- ERROR
5   end P;
6 end Volatile_Read;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Volatile_Read_1
MD5: 7ec58b4d1432d03d60b5ea6019cc031e

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
volatile_read.adb:4:12: error: volatile object in interfering context is not
↳ allowed in SPARK (SPARK RM 7.1.3(9)) [E0004]
volatile_read.adb:4:12: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳ dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Side_Effect/Volatile_
↳ Read_1/7ec58b4d1432d03d60b5ea6019cc031e/main_spark.adc:12
volatile_read.adb:4:12: error: launch "gnatprove --explain=E0004" for more
↳ information
volatile_read.adb:4:16: error: volatile object in interfering context is not
↳ allowed in SPARK (SPARK RM 7.1.3(9)) [E0004]
volatile_read.adb:4:16: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳ dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Side_Effect/Volatile_
↳ Read_1/7ec58b4d1432d03d60b5ea6019cc031e/main_spark.adc:12
volatile_read.adb:4:16: error: launch "gnatprove --explain=E0004" for more
↳ information
gnatprove: error during analysis of data and information flow

```

Instead, every read of a volatile variable must occur immediately before being assigned to another variable, as follows:

Listing 2064: volatile_read.ads

```

1 package Volatile_Read is
2   X : Integer with Volatile;

```

(continues on next page)

(continued from previous page)

```

3  procedure P (Y : out Integer);
4  end Volatile_Read;

```

Listing 2065: volatile_read.adb

```

1  package body Volatile_Read is
2      procedure P (Y : out Integer) is
3          X1 : constant Integer := X;
4          X2 : constant Integer := X;
5      begin
6          Y := X1 - X2;
7      end P;
8  end Volatile_Read;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Volatile_Read_2
MD5: 1224af597a12a8ca77b96976c76b422f

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
volatile_read.ads:3:17: info: initialization of "Y" proved

Note here that the order of capture of the volatile value of X might be significant. For example, X might denote a quantity which only increases, like clock time, so that the above expression $X1 - X2$ would always be negative or zero.

Even more significantly, functions in SPARK cannot have side effects; only procedures can. The only effect of a SPARK function is the computation of a result from its inputs, which may be passed as parameters or as global variables. In particular, SPARK functions cannot have **out** or **in out** parameters:

Listing 2066: bad_function.ads

```

1  function Bad_Function (X, Y : Integer; Sum, Max : out Integer) return Boolean;
2  -- ERROR, since "out" parameters are not allowed

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Function_With_Out_Param
MD5: 204dd22df61fe15208ae34ebc3828974

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
bad_function.ads:1:40: error: function cannot have parameter of mode "out" or "in_␣
↳out" in SPARK [E0015]
bad_function.ads:1:40: error: launch "gnatprove --explain=E0015" for more_␣
↳information
bad_function.ads:1:45: error: function cannot have parameter of mode "out" or "in_␣
↳out" in SPARK [E0015]
bad_function.ads:1:45: error: launch "gnatprove --explain=E0015" for more_␣
↳information
gnatprove: error during generation of Global contracts

More generally, SPARK does not allow functions that have a side effect in addition to returning their result, as is typical of many idioms in other languages, for example when setting

a new value and returning the previous one:

Listing 2067: bad_functions.ads

```

1 package Bad_Functions is
2   function Set (V : Integer) return Integer;
3   function Get return Integer;
4 end Bad_Functions;
```

Listing 2068: bad_functions.adb

```

1 package body Bad_Functions is
2
3   Value : Integer := 0;
4
5   function Set (V : Integer) return Integer is
6     Previous : constant Integer := Value;
7   begin
8     Value := V; -- ERROR
9     return Previous;
10  end Set;
11
12  function Get return Integer is (Value);
13
14 end Bad_Functions;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.Side_Effect_Ada
MD5: 3337b6025c4996e7fa8c7e27b4df42c1

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
bad_functions.ads:2:13: error: function "Set" with output global "Value" is not
↳ allowed in SPARK [E0005]
bad_functions.ads:2:13: error: violation of pragma SPARK_Mode at /vagrant/frontend/
↳ dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Side_Effect/Side_
↳ Effect_Ada/3337b6025c4996e7fa8c7e27b4df42c1/main_spark.adc:12
bad_functions.ads:2:13: error: launch "gnatprove --explain=E0005" for more
↳ information
gnatprove: error during analysis of data and information flow
```

GNATprove detects that function Set has a side effect on global variable Value and issues an error. The correct idiom in SPARK for such a case is to use a procedure with an **out** parameter to return the desired result:

Listing 2069: ok_subprograms.ads

```

1 package Ok_Subprograms is
2   procedure Set (V : Integer; Prev : out Integer);
3   function Get return Integer;
4 end Ok_Subprograms;
```

Listing 2070: ok_subprograms.adb

```

1 package body Ok_Subprograms is
2
3   Value : Integer := 0;
4
```

(continues on next page)

(continued from previous page)

```
5  procedure Set (V : Integer; Prev : out Integer) is
6  begin
7      Prev := Value;
8      Value := V;
9  end Set;
10
11  function Get return Integer is (Value);
12
13  end Ok_Subprograms;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Side_Effect.No_Side_Effect_Ada
MD5: 04e2235b8b6a01706434d35f6636674c

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
ok_subprograms.ads:2:32: info: initialization of "Prev" proved
ok_subprograms.ads:3:13: info: implicit aspect Always_Terminates on "Get" has been 
→ proved, subprogram will terminate

With the above restrictions in SPARK, none of the conflicts of side effects that can occur in C can occur in SPARK, and this is guaranteed by flow analysis.

DETECTING UNDEFINED BEHAVIOR

Undefined behavior (and critical unspecified behavior, which we'll treat as undefined behavior) are the plague of C programs. Many rules in MISRA C are designed to avoid undefined behavior, as evidenced by the twenty occurrences of "undefined" in the MISRA C:2012 document.

MISRA C Rule 1.3 is the overarching rule, stating very simply:

"There shall be no occurrence of undefined or critical unspecified behaviour."

The deceptive simplicity of this rule rests on the definition of *undefined or critical unspecified behaviour*. Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable.

It is therefore not surprising that a majority of MISRA C checkers do not make a serious attempt to verify compliance with MISRA C Rule 1.3.

70.1 Preventing Undefined Behavior in SPARK

Since SPARK is a subset of the Ada programming language, SPARK programs may exhibit two types of undefined behaviors that can occur in Ada:

- *bounded error*: when the program enters a state not defined by the language semantics, but the consequences are bounded in various ways. For example, reading uninitialized data can lead to a bounded error, when the value read does not correspond to a valid value for the type of the object. In this specific case, the Ada Reference Manual states that either a predefined exception is raised or execution continues using the invalid representation.
- *erroneous execution*: when when the program enters a state not defined by the language semantics, but the consequences are not bounded by the Ada Reference Manual. This is the closest to an undefined behavior in C. For example, concurrently writing through different tasks to the same unprotected variable is a case of erroneous execution.

Many cases of undefined behavior in C would in fact raise exceptions in SPARK. For example, accessing an array beyond its bounds raises the exception `Constraint_Error` while reaching the end of a function without returning a value raises the exception `Program_Error`.

The SPARK Reference Manual defines the SPARK subset through a combination of *legality rules* (checked by the compiler, or the compiler-like phase preceding analysis) and *verification rules* (checked by the formal analysis tool GNATprove). Bounded errors and erroneous execution are prevented by a combination of legality rules and the *flow analysis* part of GNATprove, which in particular detects potential reads of uninitialized data, as described in [Detecting Reads of Uninitialized Data](#) (page 1593). The following discussion focuses on how SPARK can verify that no exceptions can be raised.

70.2 Proof of Absence of Run-Time Errors in SPARK

The most common run-time errors are related to misuse of arithmetic (division by zero, overflows, exceeding the range of allowed values), arrays (accessing beyond an array bounds, assigning between arrays of different lengths), and structures (accessing components that are not defined for a given variant).

Arithmetic run-time errors can occur with signed integers, unsigned integers, fixed-point and floating-point (although with IEEE 754 floating-point arithmetic, errors are manifest as special run-time values such as NaN and infinities rather than as exceptions that are raised). These errors can occur when applying arithmetic operations or when converting between numeric types (if the value of the expression being converted is outside the range of the type to which it is being converted).

Operations on enumeration values can also lead to run-time errors; e.g., `T'Pred(T'First)` or `T'Succ(T'Last)` for an enumeration type `T`, or `T'Val(N)` where `N` is an integer value that is outside the range `0 .. T'Pos(T'Last)`.

The Update procedure below contains what appears to be a simple assignment statement, which sets the value of array element `A(I+J)` to `P/Q`.

Listing 2071: show_runtime_errors.ads

```

1 package Show_Runtime_Errors is
2
3   type Nat_Array is array (Integer range <>) of Natural;
4   -- The values in subtype Natural are 0 , 1, ... Integer'Last
5
6   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);
7
8 end Show_Runtime_Errors;
```

Listing 2072: show_runtime_errors.adb

```

1 package body Show_Runtime_Errors is
2
3   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
4   begin
5     A (I + J) := P / Q;
6   end Update;
7
8 end Show_Runtime_Errors;
```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Undefined_Behavior.Runtime_Errors
MD5: 8ad4488974ab9e49ac17bf094ae33eac

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
show_runtime_errors.adb:5:12: medium: overflow check might fail, cannot prove
↳ lower bound for I + J [reason for check: result of addition must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (if J >= 0 then I <=
↳ Integer'Last - J else I >= Integer'First - J) to subprogram at show_runtime_
↳ errors.ads:6]
show_runtime_errors.adb:5:12: medium: array index check might fail [reason for
↳ check: result of addition must be a valid index into the array] [possible fix:
↳ add precondition (if J >= 0 then I <= A'Last - J else I >= A'First - J) to
↳ subprogram at show_runtime_errors.ads:6]
```

(continues on next page)

(continued from previous page)

```

show_runtime_errors.adb:5:22: medium: divide by zero might fail [possible fix: add
↳ precondition (Q /= 0) to subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:22: medium: overflow check might fail, cannot prove
↳ lower bound for P / Q [reason for check: result of division must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (P / Q >= Integer'First)
↳ to subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:22: medium: range check might fail, cannot prove lower
↳ bound for P / Q [reason for check: result of division must fit in the target
↳ type of the assignment] [possible fix: add precondition (P / Q >= Natural'First)
↳ to subprogram at show_runtime_errors.ads:6]
gnatprove: unproved check messages considered as errors

```

However, for an arbitrary invocation of this procedure, say `Update(A, I, J, P, Q)`, an exception can be raised in a variety of circumstances:

- The computation `I+J` may overflow, for example if `I` is `Integer'Last` and `J` is positive.

```
A (Integer'Last + 1) := P / Q;
```

- The value of `I+J` may be outside the range of the array `A`.

```
A (A'Last + 1) := P / Q;
```

- The division `P / Q` may overflow in the special case where `P` is `Integer'First` and `Q` is `-1`, because of the asymmetric range of signed integer types.

```
A (I + J) := Integer'First / -1;
```

- Since the array can only contain non-negative numbers (the element subtype is `Natural`), it is also an error to store a negative value in it.

```
A (I + J) := 1 / -1;
```

- Finally, if `Q` is `0` then a divide by zero error will occur.

```
A (I + J) := P / 0;
```

For each of these potential run-time errors, the compiler will generate checks in the executable code, raising an exception if any of the checks fail:

```

A (Integer'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
-- raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
-- raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
-- raised CONSTRAINT_ERROR : divide by zero

```

These run-time checks incur an overhead in program size and execution time. Therefore it may be appropriate to remove them if we are confident that they are not needed.

The traditional way to obtain the needed confidence is through testing, but it is well known that this can never be complete, at least for non-trivial programs. Much better is to guarantee the absence of run-time errors through sound static analysis, and that's where SPARK and GNATprove can help.

More precisely, GNATprove logically interprets the meaning of every instruction in the program, taking into account both control flow and data/information dependencies. It uses this analysis to generate a logical formula called a *verification condition* for each possible check.

```
A (Integer'Last + 1) := P / Q;  
-- medium: overflow check might fail  
  
A (A'Last + 1) := P / Q;  
-- medium: array index check might fail  
  
A (I + J) := Integer'First / (-1);  
-- medium: overflow check might fail  
  
A (I + J) := 1 / (-1);  
-- medium: range check might fail  
  
A (I + J) := P / 0;  
-- medium: divide by zero might fail
```

The verification conditions are then given to an automatic prover. If every verification condition can be proved, then no run-time errors will occur.

GNATprove's analysis is sound — it will detect all possible instances of run-time exceptions being raised — while also having high precision (i.e., not producing a cascade of "false alarms").

The way to program in SPARK so that GNATprove can guarantee the absence of run-time errors entails:

- declaring variables with precise constraints, and in particular to specify precise ranges for scalars; and
- defining preconditions and postconditions on subprograms, to specify respectively the constraints that callers should respect and the guarantees that the subprogram should provide on exit.

For example, here is a revised version of the previous example, which can guarantee through proof that no possible run-time error can be raised:

Listing 2073: no_runtime_errors.ads

```
1 package No_Runtime_Errors is  
2  
3   subtype Index_Range is Integer range 0 .. 100;  
4  
5   type Nat_Array is array (Index_Range range <>) of Natural;  
6  
7   procedure Update (A : in out Nat_Array;  
8                   I, J : Index_Range;  
9                   P, Q : Positive)  
10  with  
11    Pre => I + J in A'Range;  
12  
13 end No_Runtime_Errors;
```

Listing 2074: no_runtime_errors.adb

```
1 package body No_Runtime_Errors is  
2  
3   procedure Update (A : in out Nat_Array;  
4                   I, J : Index_Range;  
5                   P, Q : Positive) is
```

(continues on next page)

(continued from previous page)

```
6   begin
7     A (I + J) := P / Q;
8   end Update;
9
10  end No_Runtime_Errors;
```


DETECTING UNREACHABLE CODE AND DEAD CODE

MISRA C defines *unreachable code* as code that cannot be executed, and it defines *dead code* as code that can be executed but has no effect on the functional behavior of the program. (These definitions differ from traditional terminology, which refers to the first category as "dead code" and the second category as "useless code".) Regardless of the terminology, however, both types are actively harmful, as they might confuse programmers and lead to errors during maintenance.

The "Unused code" section of MISRA C contains seven rules that deal with detecting both unreachable code and dead code. The two most important rules are:

- Rule 2.1: "A project shall not contain unreachable code", and
- Rule 2.2: "There shall not be dead code".

Other rules in the same section prohibit unused entities of various kinds (type declarations, tag declarations, macro declarations, label declarations, function parameters).

While some simple cases of unreachable code can be detected by static analysis (typically if a condition in an **if** statement can be determined to be always true or false), most cases of unreachable code can only be detected by performing coverage analysis in testing, with the caveat that code reported as not being executed is not necessarily unreachable (it could simply reflect gaps in the test suite). Note that *statement coverage*, rather than the more comprehensive *decision coverage* or *modified condition / decision coverage* (MC/DC) as defined in the DO-178C standard for airborne software, is sufficient to detect potential unreachable statements, corresponding to code that is not covered during the testing campaign.

The presence of dead code is much harder to detect, both statically and dynamically, as it requires creating a complete dependency graph linking statements in the code and their effect on visible behavior of the program.

SPARK can detect some cases of both unreachable and dead code through its precise construction of a dependency graph linking a subprogram's statements to all its inputs and outputs. This analysis might not be able to detect complex cases, but it goes well beyond what other analyses do in general.

Listing 2075: much_ado_about_little.ads

```
1  procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean);
```

Listing 2076: much_ado_about_little.adb

```
1  procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean) is
2
3      procedure Ok is
4      begin
5          Success := True;
6      end Ok;
7
```

(continues on next page)

(continued from previous page)

```

8   procedure NOK is
9   begin
10      Success := False;
11  end NOK;
12
13 begin
14   Success := False;
15
16   for K in Y .. Z loop
17     if K < X and not Success then
18       Ok;
19     end if;
20   end loop;
21
22   if X > Y then
23     Ok;
24   else
25     NOK;
26   end if;
27
28   if Z > Y then
29     NOK;
30     return;
31   else
32     Ok;
33     return;
34   end if;
35
36   if Success then
37     Success := not Success;
38   end if;
39 end Much_Ado_About_Little;

```

Code block metadata

Project: Courses.SPARK_For_The_MISRA_C_Dev.Unreachable_And_Dead_Code.Much_Ado_About_Little
 MD5: ccccb112fbab169ba964b3f8ef36ec2d

Build output

much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]

Prover output

Phase 1 of 3: generation of data representation information ...
 Phase 2 of 3: generation of Global contracts ...
 Phase 3 of 3: analysis of data and information flow ...
 much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at ↵
 ↵much_ado_about_little.adb:18
 much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at ↵
 ↵much_ado_about_little.adb:23
 much_ado_about_little.adb:10:15: warning: unused assignment, in call inlined at ↵
 ↵much_ado_about_little.adb:25
 much_ado_about_little.adb:14:12: warning: unused assignment
 much_ado_about_little.adb:16:20: warning: statement has no effect
 much_ado_about_little.adb:17:07: warning: statement has no effect
 much_ado_about_little.adb:22:04: warning: statement has no effect
 much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]
 much_ado_about_little.adb:36:04: warning: this statement is never reached
 much_ado_about_little.adb:37:15: warning: this statement is never reached

(continues on next page)

(continued from previous page)

```
much_ado_about_little.ads:1:34: warning: unused initial value of "X"
```

The only code in the body of `Much_Ado_About_Little` that affects the result of the procedure's execution is the `if Z > Y...` statement, since this statement sets `Success` to either `True` or `False` regardless of what the previous statements did. I.e., the statements preceding this `if` are dead code in the MISRA C sense. Since both branches of the `if Z > Y...` statement return from the procedure, the subsequent `if Success...` statement is unreachable. GNATprove detects and issues warnings about both the dead code and the unreachable code.

CONCLUSION

The C programming language is "close to the metal" and has emerged as a *lingua franca* for the majority of embedded platforms of all sizes. However, its software engineering deficiencies (such as the absence of data encapsulation) and its many traps and pitfalls present major obstacles to those developing critical applications. To some extent, it is possible to put the blame for programming errors on programmers themselves, as Linus Torvalds admonished:

"Learn C, instead of just stringing random characters together until it compiles (with warnings)."

But programmers are human, and even the best would be hard pressed to be 100% correct about the myriad of semantic details such as those discussed in this document. Programming language abstractions have been invented precisely to help developers focus on the "big picture" (thinking in terms of problem-oriented concepts) rather than low-level machine-oriented details, but C lacks these abstractions. As Kees Cook from the Kernel Self Protection Project puts it (during the Linux Security Summit North America 2018):

"Talking about C as a language, and how it's really just a fancy assembler"

Even experts sometimes have problems with the C programming language rules, as illustrated by Microsoft expert David LeBlanc (see [Enforcing Strong Typing for Scalars](#) (page 1583)) or the MISRA C Committee itself (see the [Preface](#) (page 1553)).

The rules in MISRA C represent an impressive collective effort to improve the reliability of C code in critical applications, with a focus on avoiding error-prone features rather than enforcing a particular programming style. The Rationale provided with each rule is a clear and unobjectionable justification of the rule's benefit.

At a fundamental level, however, MISRA C is still built on a base language that was not really designed with the goal of supporting large high-assurance applications. As shown in this document, there are limits to what static analysis can enforce with respect to the MISRA C rules. It's hard to retrofit reliability, safety and security into a language that did not have these as goals from the start.

The SPARK language took a different approach, starting from a base language (Ada) that was designed from the outset to support solid software engineering, and eliminating features that were implementation dependent or otherwise hard to formally analyze. In this document we have shown how the SPARK programming language and its associated formal verification tools can contribute usefully to the goal of producing error-free software, going beyond the guarantees that can be achieved in MISRA C.

REFERENCES

73.1 About MISRA C

The official website of the MISRA association <https://www.misra.org.uk/> has many freely available resources about MISRA C, some of which can be downloaded after registering on the MISRA Bulletin Board at <https://www.misra.org.uk/forum/> (such as the examples from the MISRA C:2012 standard, which includes a one-line description of each guideline).

The following documents are freely available:

- *MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines*, 2016, which explains the rationale and process for compliance, including a thorough discussion of acceptable deviations
- *MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012*, 2016, which contains 14 additional guidelines focusing on security. This is a minor addition to MISRA C.

The main MISRA C:2012 document can be purchased from the MISRA webstore.

PRQA is the company that first developed MISRA C, and they have been heavily involved in every version since then. Their webpage <http://www.prqa.com/coding-standards/misra/> contains many resources about MISRA C: product datasheets, white papers, webinars, professional courses.

The PRQA Resources Library at http://info.prqa.com/resources-library?filter=white_paper has some freely available white papers on MISRA C and the use of static analyzers:

- An introduction to MISRA C:2012 at http://info.prqa.com/MISRA_C-2012-whitepaper-evaluation-lp
- *The Myth of Perfect MISRA Compliance* at <http://info.prqa.com/myth-of-perfect-MISRA-Compliance-evaluation-lp>, providing background information on the use and limitations of static analyzers for checking MISRA C compliance

In 2013 ISO standardized a set of 45 rules focused on security, available in the *C Secure Coding Rules*. A draft is freely available at <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf>

In 2018 MISRA published *MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure"*, mapping ISO rules to MISRA C:2012 guidelines. This document is freely available from <https://www.misra.org.uk/>.

73.2 About SPARK

The e-learning website <https://learn.adacore.com/> contains a freely available interactive course on SPARK.

The SPARK User's Guide is available at <http://docs.adacore.com/spark2014-docs/html/ug/>.

The SPARK Reference Manual is available at <http://docs.adacore.com/spark2014-docs/html/lrm/>.

A student-oriented textbook on SPARK is *Building High Integrity Applications with SPARK* by John McCormick and Peter Chapin, published by Cambridge University Press. It covers the latest version of the language, SPARK 2014.

A historical account of the evolution of SPARK technology and its use in industry is covered in the article *Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK* by Roderick Chapman and Florian Schanda, at <http://proteancode.com/keynote.pdf>

The website <https://www.adacore.com/sparkpro> is a portal for up-to-date information and resources on SPARK. AdaCore blog's site <https://blog.adacore.com/> contains a number of SPARK-related posts.

The booklet *AdaCore Technologies for Cyber Security* shows how AdaCore's technology can be used to prevent or mitigate the most common security vulnerabilities in software. See <https://www.adacore.com/books/adacore-tech-for-cyber-security/>.

The booklet *AdaCore Technologies for CENELEC EN 50128:2011* shows how AdaCore's technology can be used in conjunction with the CENELEC EN 50128:2011 software standard for railway control and protection systems. It describes in particular where the SPARK technology fits best and how it can be used to meet various requirements of the standard. See: <https://www.adacore.com/books/cenelec-en-50128-2011/>.

The booklet *AdaCore Technologies for DO-178C/ED-12C* similarly shows how AdaCore's technology can be used in conjunction with the DO-178C/ED-12C standard for airborne software, and describes in particular how SPARK can be used in conjunction with the Formal Methods supplement DO-333/ED-216. See <https://www.adacore.com/books/do-178c-tech/>.

73.3 About MISRA C and SPARK

The blog post at <https://blog.adacore.com/MISRA-C-2012-vs-spark-2014-the-subset-matching-game> reviews the 27 undecidable rules in MISRA C:2012 and describes how SPARK addresses them.

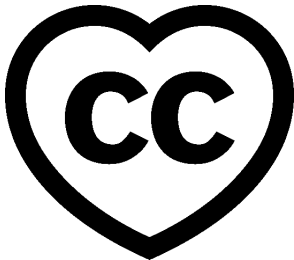
The white paper *A Comparison of SPARK with MISRA C and Frama-C* at <https://www.adacore.com/papers/compare-spark-MISRA-C-frama-c> compares SPARK to MISRA C and to the formal verification tool Frama-C for C programs.

Part VII

Introduction to Embedded Systems Programming

Copyright © 2022 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page⁴²²](https://creativecommons.org/licenses/by-sa/4.0)



This course will teach you the basics of the Embedded Systems Programming using Ada.

This document was written by Patrick Rogers, with review by Stephen Baird, Tucker Taft, Filip Gajowniczek, and Gustavo A. Hoffmann.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

⁴²² <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

This is a course about embedded systems programming. Embedded systems are everywhere today, including — just to name a few — the thermostats that control a building's temperature, the power-steering controller in modern automobiles, and the control systems in charge of jet engines.

Clearly, much can depend on these systems operating correctly. It might be only a matter of comfort if the thermostat fails. But imagine what might happen if one of the critical control systems in your car failed when you're out on the freeway. When a jet engine controller is designed to have absolute control, it is known as a Full Authority Digital Engine Controller, or FADEC for short. If a FADEC fails, the result can make international news.

Using Ada can help you get it right, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal is to get you started in using it well.

The course is based on the assumption that you know some of the Ada language already, preferably even some of the more advanced concepts. You don't need to know how to use Ada constructs for embedded systems, of course, but you do need to know at least the language basics. If you need that introduction, see the course [Introduction to Ada](#) (page 5).

We also assume that you already have some programming experience so we won't cover CS-101.

Ideally, you also have some experience with low-level programming, because we will focus on "how to do it in Ada." If you do, feel free to gloss over the introductory material. If not, don't worry. We will cover enough for the course to be of value in any case.

74.1 So, what will we actually cover?

We will introduce you to using Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data, e.g., data from sensors that are occasionally faulty.

We will discuss development using more than Ada alone, nowadays a quite common approach. Specifically, how to interface with code and data written in other languages, and how (and why) to work with assembly language.

Embedded systems interact with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. Frequently these devices are mapped into the target memory address space. We will cover how to define and interact with these memory-mapped devices.

Finally, we will show how to handle interrupts in Ada, using portable constructs.

74.2 Definitions

Before we go any further, what do we mean by "embedded system" anyway? It's time to be specific. We're talking about a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system: the enclosing thermostat controlling the temperature, the power steering controller in the enclosing automobile, and the FADEC embedded in the enclosing aircraft. So these are not stand-alone computers for general purpose application execution.

As such, embedded systems typically have reduced resources available, especially power, which means reduced processor speed and reduced memory on-board. For an example at the small end of the spectrum, consider the computer embedded in a wearable device: it must run for a long time on a very little battery, with comparatively little memory available. But that's often true of bigger systems too, such as systems on aircraft where power (and heat) are directly limiting factors.

As a result, developing embedded systems software can be more difficult than general application development, not to mention that this software is potentially safety-critical.

Ada is known for use in very large, very long-lived projects (e.g., deployed for decades), but it can also be used for very small systems with tight resource constraints. We'll show you how.

We used the term "computer" above. You already know what that means, but you may be thinking of your laptop or something like that, where the processor, memory, and devices are all distinct, separate components. That can be the case for embedded systems too, albeit in a different form-factor such as rack-mounted boards. However, be sure to expand your definition to include the notion of a system-on-chip (SoC), in which the processor, memory, and various useful devices are all on a single chip. Embedded systems don't necessarily involve SoC computers but they frequently do. The techniques and information in this course work on any of these kinds of computer.

74.3 Down To The Bare Metal

Ada has always had facilities designed specifically for embedded systems. The language includes constructs for directly manipulating hardware, for example, and direct interaction with assembly language. These constructs are as effective as those of any high-level programming language (yes, including C). These constructs are expressively powerful, well-specified (so there are few surprises), efficient, and portable (within reason).

We say "within reason" because portability is a difficult goal for embedded systems. That's because the hardware is so much a part of the application itself, rather than being abstracted away as in a general-purpose application. That said, the hardware details can be managed in Ada so that portability is maximized to the extent possible for the application.

But strictly speaking, not all software can or should be absolutely portable! If a specific device is required, well, the program won't work with some other device. But to the extent possible portability is obviously a good thing.

74.4 The Ada Drivers Library

Speaking of SoC computers, there is a library of freely-available device drivers in Ada. Known as the Ada Driver Library (ADL), it supports many devices on a number of vendors' products. Device drivers for timers, I2C, SPI, A/D and D/A converters, DMA, General Purpose I/O, LCD displays, sensors, and other devices are included. The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

An extensive description of a project using the ADL is available here: <https://blog.adacore.com/making-an-rc-car-with-ada-and-spark>

We will refer to components of this library and use some of them as examples.

LOW LEVEL PROGRAMMING

This section introduces a number of topics in low-level programming, in which the hardware and the compiler's representation choices are much more in view at the source code level. In comparatively high level code these topics are "abstracted away" in that the programmer can assume that the compiler does whatever is necessary on the current target machine so that their code executes as intended. That approach is not sufficient in low-level programming.

Note that we do not cover every possibility or language feature. Instead, we cover the necessary concepts, and also potential surprises or pitfalls, so that the parts not covered can be learned on your own.

75.1 Separation Principle

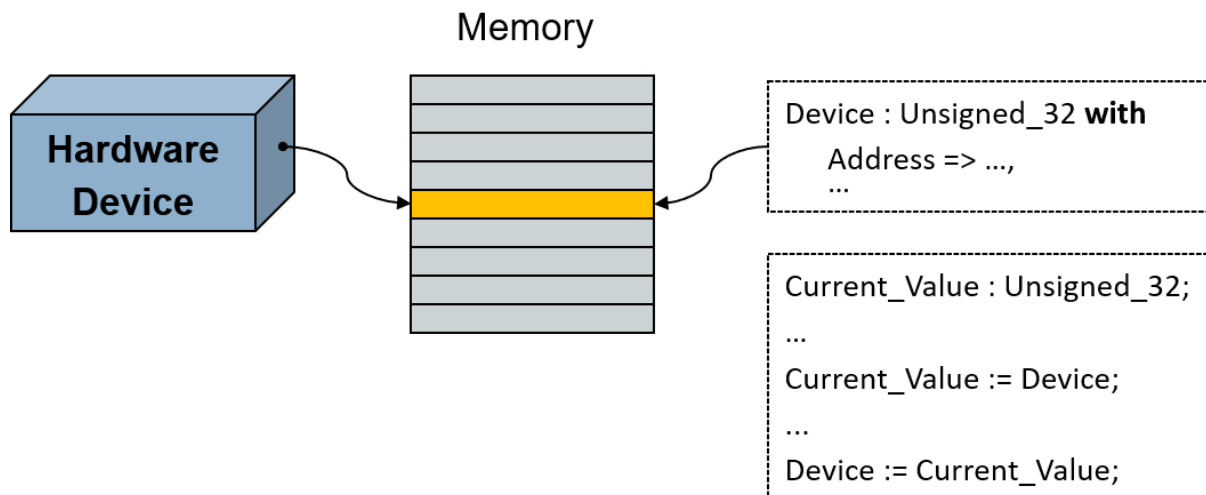
There is a language design principle underlying the Ada facilities intended for implementing embedded software. This design principle directly affects how the language is used, and therefore, the portability and readability of the resulting application code.

This language design principle is known as the "separation principle." What's being separated? The low-level, less portable aspects of some piece of code are separated from the usage of that piece of code.

Don't confuse this with hiding unnecessary implementation details via compile-time visibility control (i.e., information hiding and encapsulation). That certainly should be done too. Instead, because of the separation principle, we specify the low-level properties of something once, when we declare it. From then on, we can use regular Ada code to interact with it. That way the bulk of the code — the usage — is like any other Ada code, and doesn't propagate the low-level details all over the client code. This greatly simplifies usage and understandability as well as easing porting to new hardware-specific aspects. You change things in one place, rather than everywhere.

For example, consider a device mapped to the memory address space of the processor. To interact with the device we interact with one or more memory cells. Reading input from the device amounts to reading the value at the associated memory location. Likewise, sending output to the device amounts to writing to that location.

To represent this device mapping we declare a variable of an appropriate type and specify the starting address the object should occupy. (There are other ways too, but for a single, statically mapped object this is the simplest approach.) We'd want to specify some other characteristics as well, but let's focus on the address.



If the hardware presents an interface consisting of multiple fields within individual memory cells, we can use a record type instead of a single unsigned type representing a single word. Ada allows us to specify the exact record layout, down to the individual bit level, for any types we may need to use for the record components. When we declare the object we use that record type, again specifying the starting address. Then we can just refer to the object's record components as usual, having the compiler compute the address offsets required to access the components representing the individual hardware fields.

Note that we aren't saying that other languages cannot do this too. Many can, using good programming practices. What we're saying is that those practices are designed into the Ada way of doing it.

75.2 Guaranteed Level of Support

The Ada reference manual has an entire section dedicated to low-level programming. That's section 13, "Representation Issues," which provides facilities for developers to query and control aspects of various entities in their code, and for interfacing to hardware. Want to specify the exact layout for a record type's components? Easy, and the compiler will check your layout too. Want to specify the alignment of a type? That's easy too. And that's just the beginning. We'll talk about these facilities as we go, but there's another point to make about this section.

In particular, section 13 includes recommended levels of support to be provided by language implementations, i.e., compilers and other associated tools. Although the word "recommended" is used, the recommendations are meant to be followed.

For example, section 13.3 says that, for some entity named *X*, "*X*' [Address](#) should produce a useful result if *X* is an object that is aliased or of a by-reference type, or is an entity whose [Address](#) has been specified." So, for example, if the programmer specifies the address for a memory-mapped variable, the compiler cannot ignore that specification and instead, for the sake of performance, represent that variable using a register. The object must be represented as an addressable entity, as requested by the programmer. (Registers are not addressable.)

We mention this because, although the recommended levels of support are intended to be followed, those recommendations become **requirements** if the Systems Programming (SP) Annex is implemented by the vendor. In that case the vendor's implementation of section 13 must support at least the recommended levels. The SP Annex defines additional, optional functionality oriented toward this programming domain; you want it anyway. (Like all the annexes it adds no new syntax.) Almost all vendors, if not literally all, implement the Annex so you can rely on the recommended levels of support.

75.3 Querying Implementation Limits and Characteristics

Sometimes you need to know more about the underlying machine than is typical for general purpose applications. For example, your numerical analysis algorithm might need to know the maximum number of digits of precision that a floating-point number can have on this specific machine. For networking code, you will need to know the "endianness" of the machine so you can know whether to swap the bytes in an Ethernet packet. You'd go look in the `limits.h` file in C implementations, but in Ada we go to a package named `System` to get this information.

Clearly, these implementation values will vary with the hardware, so the package declares constants with implementation-defined values. The names of the constants are what's portable, you can count on them being the same in any Ada implementation.

However, vendors can add implementation-defined declarations to the language-defined content in package `System`. You might require some of those additions, but portability could then suffer when moving to a new vendor's compiler. Try not to use them unless it is unavoidable. Ideally these additions will appear in the private part of the package, so the implementation can use them but application code cannot.

For examples of the useful, language-defined constants, here are those for the numeric limits of an Ada compiler for an Arm 32-bit SoC:

```
Min_Int      : constant := Long_Long_Integer'First;
Max_Int      : constant := Long_Long_Integer'Last;

Max_Binary_Modulus : constant := 2 ** Long_Long_Integer'Size;
Max_Nonbinary_Modulus : constant := 2 ** Integer'Size - 1;

Max_Base_Digits : constant := Long_Long_Float'Digits;
Max_Digits      : constant := Long_Long_Float'Digits;

Max_Mantissa    : constant := 63;
Fine_Delta      : constant := 2.0 ** (-Max_Mantissa);
```

`Min_Int` and `Max_Int` supply the most-negative and most-positive integer values supported by the machine.

`Max_Binary_Modulus` is the largest power of two allowed as the modulus of a modular type definition.

But a modular type need not be defined in terms of powers of two. An arbitrary modulus is allowed, as long as it is not bigger than the machine can handle. That's specified by `Max_Nonbinary_Modulus`, the largest non-power-of-two value allowed as the modulus of a modular type definition.

`Max_Base_Digits` is the largest value allowed for the requested decimal precision in a floating-point type's definition.

We won't go over all of the above, you get the idea. Let's examine the more important contents.

Two of the most frequently referenced constants in `System` are the following, especially the first. (The values here are again for the Arm 32-bit SoC):

```
Storage_Unit : constant := 8;
Word_Size    : constant := 32;
```

`Storage_Unit` is the number of bits per memory storage element. Storage elements are the components of memory cells, and typically correspond to the individually addressable memory elements. A "byte" would correspond to a storage element with the above constant value.

Consider a typical idiom for determining the number of whole storage elements an object named *X* occupies:

```
Units : constant Integer := (X'Size + Storage_Unit - 1) / Storage_Unit;
```

Remember that *'Size* returns a value in terms of bits. There are more direct ways to determine that size information but this will serve as an example of the sort of thing you might do with that constant.

A machine "word" is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of some number of storage elements, maybe one but typically more than one. As the unit the machine natively manipulates, words are expected to be independently addressable. (On some machines only words are independently addressable.)

Word_Size is the number of bits in the machine word. On a 32-bit machine we'd expect *Word_Size* to have a value of 32; on a 64-bit machine it would probably be 64, and so on.

Storage_Unit and *Word_Size* are obviously related.

Another frequently referenced declaration in package *System* is that of the type representing memory addresses, along with a constant for the null address designating no storage element.

```
type Address is private;  
Null_Address : constant Address;
```

You may be wondering why type *Address* is a private type, since that choice means that we programmers cannot treat it like an ordinary (unsigned) integer value. Portability is of course the issue, because addressing, and thus address representation, varies among computer architectures. Not all architectures have a flat address space directly referenced by numeric values, although that is common. Some are represented by a base address plus an offset, for example. Therefore, the representation for type *Address* is hidden from us, the clients. Consequently we cannot simply treat address values as numeric values. Don't worry, though. The operations we need are provided.

Package *System* declares these comparison functions, for example:

```
function "<" (Left, Right : Address) return Boolean;  
function "<=" (Left, Right : Address) return Boolean;  
function ">" (Left, Right : Address) return Boolean;  
function ">=" (Left, Right : Address) return Boolean;  
function "=" (Left, Right : Address) return Boolean;
```

These functions are intrinsic, i.e., built-in, meaning that the compiler generates the code for them directly at the point of calls. There is no actual function body for any of them so there is no performance penalty.

Any private type directly supports the equality function, and consequently the inequality function, as well as assignment. What we don't get here is address arithmetic, again because we don't have a compile-time view of the actual representation. That functionality is provided by package *System.Storage_Elements*, a child package we will cover later. We should say though, that the need for address arithmetic in Ada is rare, especially compared to C.

Having type *Address* presented as a private type is not, strictly speaking, required by the language. Doing so is a good idea for the reasons given above, and is common among vendors. Not all vendors do, though.

Note that *Address* is the type of the result of the query attribute *Address*.

We mentioned potentially needing to swap bytes in networking communications software, due to the differences in the "endianness" of the machines communicating. That characteristic can be determined via a constant declared in package *System* as follows:

```

type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order := implementation-defined;

```

High_Order_First corresponds to "Big Endian" and Low_Order_First to "Little Endian." On a Big Endian machine, bit 0 is the most significant bit. On a Little Endian machine, bit 0 is the least significant bit.

Strictly speaking, this constant gives us the default order for bits within storage elements in record representation clauses, not the order of bytes within words. However, we can usually use it for the byte order too. In particular, if Word_Size is greater than Storage_Unit, a word necessarily consists of multiple storage elements, so the default bit ordering is the same as the ordering of storage elements in a word.

Let's take that example of swapping the bytes in a received Ethernet packet. The "wire" format is Big Endian so if we are running on a Little Endian machine we must swap the bytes received.

Suppose we want to retrieve typed values from a given buffer of bytes. We get the bytes from the buffer into a variable named Value, of the type of interest, and then swap those bytes within Value if necessary.

```

...
begin
  Value := ...

  if Default_Bit_Order /= High_Order_First then
    -- we're not on a Big Endian machine
    Value := Byte_Swapped (Value);
  end if;
end Retrieve_4_Bytes;

```

We have elided the code that gets the bytes into Value, for the sake of simplicity. How the bytes are actually swapped by function Byte_Swapped is also irrelevant. The point here is the if-statement: the expression compares the Default_Bit_Order constant to High_Order_First to see if this execution is on a Big Endian machine. If not, it swaps the bytes because the incoming bytes are always received in "wire-order," i.e., Big Endian order.

Another important set of declarations in package System define the values for priorities, including interrupt priorities. We will ignore them until we get to the section on interrupt handling.

Finally, and perhaps surprisingly, a few declarations in package System are almost always (if not actually always) ignored.

```

type Name is implementation-defined-enumeration-type;
System_Name : constant Name := implementation-defined;

```

Values of type Name are the names of alternative machine configurations supported by the implementation. System_Name represents the current machine configuration. We've never seen any actual use of this.

Memory_Size is an implementation-defined value that is intended to reflect the memory size of the configuration, in units of storage elements. What the value actually refers to is not specified. Is it the size of the address space, i.e., the amount possible, or is it the amount of physical memory actually on the machine, or what? In any case, the amount of memory available to a given computer is neither dependent upon, nor reflected by, this constant. Consequently, Memory_Size is not useful either.

Why have something defined in the language that nobody uses? In short, it seemed like a good idea at the time when Ada was first defined. Upward-compatibility concerns propagate these declarations forward as the language evolves, just in case somebody does use them.

75.4 Querying Representation Choices

As we mentioned in the introduction, in low-level programming the hardware and the compiler's representation choices can come to the forefront. You can, therefore, query many such choices.

For example, let's say we want to query the addresses of some objects because we are calling the imported C memcpy function. That function requires two addresses to be passed to the call: one for the source, and one for the destination. We can use the 'Address attribute to get those values.

We will explore importing routines and objects implemented in other languages elsewhere. For now, just understand that we will have an Ada declaration for the imported routine that tells the compiler how it should be called. Let's assume we have an Ada function declared like so:

```
function MemCopy
  (Destination : System.Address;
   Source      : System.Address;
   Length      : Natural)
return Address
with
  Import,
  Convention => C,
  Link_Name => "memcpy",
  Pre  => Source /= Null_Address and then
        Destination /= Null_Address and then
        not Overlapping (Destination, Source, Length),
  Post => MemCopy'Result = Destination;
-- Copies Length bytes from the object designated by Source to the object
-- designated by Destination.
```

The three aspects that do the importing are specified after the reserved word **with** but can be ignored for this discussion. We'll talk about them later. The preconditions make explicit the otherwise implicit requirements for the arguments passed to memcpy, and the postcondition specifies the expected result returned from a successful call. Neither the preconditions nor the postconditions are required for importing external entities but they are good "guard-rails" for using those entities. If we call it incorrectly the precondition will inform us, and likewise, if we misunderstand the result the postcondition will let us know (at least to the extent that the return value does that).

For a sample call to our imported routine, imagine that we have a procedure that copies the bytes of a **String** parameter into a Buffer parameter, which is just a contiguous array of bytes. We need to tell MemCopy the addresses of the arguments passed so we apply the 'Address attribute accordingly:

```
procedure Put (This : in out Buffer; Start : Index; Value : String) is
  Result : System.Address with Unreferenced;
begin
  Result := MemCopy (Destination => This (Start)'Address,
                    Source      => Value'Address,
                    Length      => Value'Length);
end Put;
```

The order of the address parameters is easily confused so we use the named association format for specifying the actual parameters in the call.

Although we assign Result we don't otherwise use it, so we tell the compiler this is not a mistake via the Unreferenced aspect. And if we do turn around and reference it the compiler will complain, as it should. Note that Unreferenced is defined by GNAT, so usage is not necessarily portable. Other vendors may or may not implement something like it, perhaps with a different name.

(We don't show the preconditions for `Put`, but they would have specified that `Start` must be a valid index into this particular buffer, and that there must be room in the `Buffer` argument for the number of bytes in `Value` when starting at the `Start` index, so that we don't copy past the end of the `Buffer` argument.)

There are other characteristics we might want to query too.

We might want to ask the compiler what alignment it chose for a given object (or type, for all such objects).

For a type, when `Alignment` returns a non-zero value we can be sure that the compiler will allocate storage for objects of the type at correspondingly aligned addresses (unless we force it to do otherwise). Similarly, references to dynamically allocated objects of the type will be to properly aligned locations. Otherwise, an `Alignment` of zero means that the guarantee does not hold. That could happen if the type is packed down into a composite object, such as an array of `Booleans`. We'll discuss "packing" soon. More commonly, the smallest likely value is 1, meaning that any storage element's address will suffice. If the machine has no particular natural alignments, then all type alignments will probably be 1 by default. That would be somewhat rare today, though, because modern processors usually have comparatively strict alignment requirements.

We can ask for the amount of storage associated with various entities. For example, when applied to a task, '`Storage_Size`' tells us the number of storage elements reserved for the task's execution. The value includes the size of the task's stack, if it has one. We aren't told if other required storage, used internally in the implementation, is also included in this number. Often that other storage is not included in this number, but it could be.

`Storage_Size` is also defined for access types. The meaning is a little complicated. Access types can be classified into those that designate only variables and constants ("access-to-object") and those that can designate subprograms. Each access-to-object type has an associated storage pool. The storage allocated by `new` comes from the pool, and instances of `Unchecked_Deallocation` return storage to the pool.

When applied to an access-to-object type, `Storage_Size` gives us the number of storage elements reserved for the corresponding pool.

Note that `Storage_Size` doesn't tell us how much available, unallocated space remains in a pool. It includes both allocated and unallocated space. Note, too, that although each access-to-object type has an associated pool, that doesn't mean that each one has a distinct, dedicated pool. They might all share one, by default. On an operating system, such as Linux, the default shared pool might even be implicit, consisting merely of calls to the OS routines in C.

As a result, querying `Storage_Size` for access types and tasks is not necessarily all that useful. Specifying the sizes, on the other hand, definitely can be useful.

That said, we can create our own pool types and define precisely how they are sized and how allocation and deallocation work, so in that case querying the size for access types could be more useful.

For an array type or object, '`Component_Size`' provides the size in bits of the individual components.

More useful are the following two attributes that query a degree of memory sharing between objects.

Applied to an object, '`Has_Same_Storage`' is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations occupy exactly the same bits.

Applied to an object, '`Overlaps_Storage`' is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations share at least one bit.

Generally, though, we specify representation characteristics far more often than we query them. Rather than describe all the possibilities, we can just say that all the representation characteristics that can be specified can also be queried. We cover specifying representation characteristics next, so just assume the corresponding queries are available.

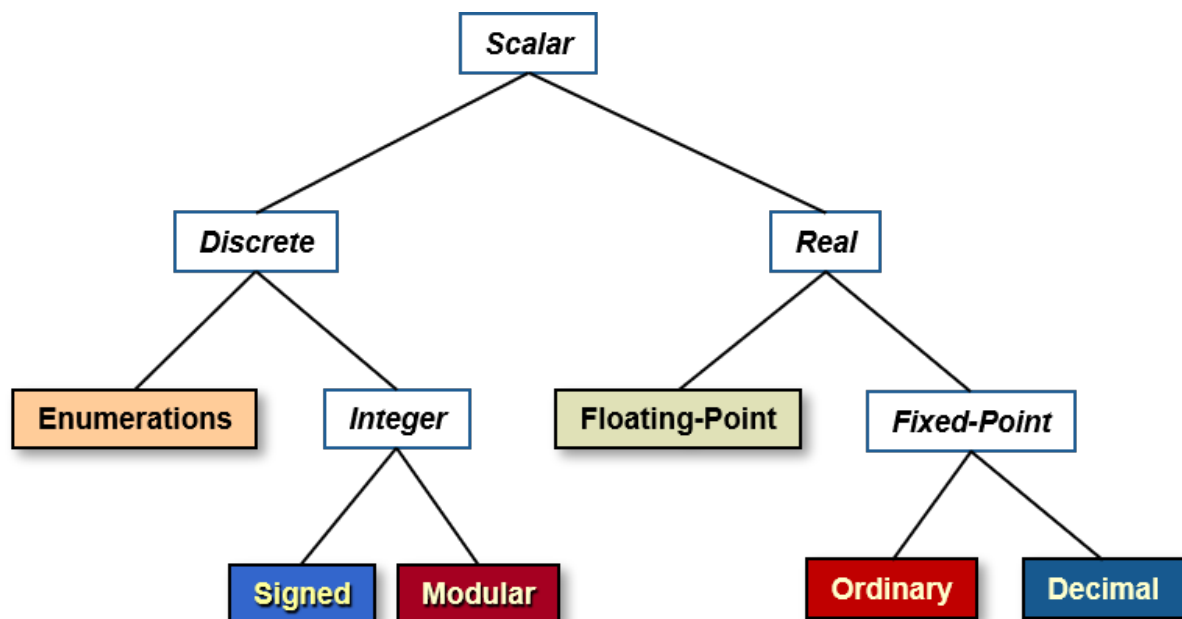
That said, there is one particular representation query we need to talk about explicitly, now, because there is a lot of confusion about it: the `'Size` attribute. The confusion stems from the fact that there are multiple contexts for applying the attribute, and multiple reasonable interpretations possible. We can apply the `'Size` attribute to a type, in an attempt to get information about all objects of the type, or we can apply it to individual objects to get specific information. In both cases, what actual information do we get? In the original version of Ada these questions weren't really answered so vendors did what they thought was correct. But they did not agree with each other, and portability became a problem.

For example, suppose you want to convert some value to a series of bytes in order to send the value over the wire. To do that you need to know how many bytes are required to represent the value. Many applications queried the size of the type to determine that, and then, when porting to a new vendor's compiler, found that their code no longer worked correctly. The new vendor's implementation wasn't wrong, it was just different.

Later versions of Ada answered these questions, where possible, so let's examine the contexts and meaning. Above all, though, remember that `'Size` returns values in terms of **bits**.

If we apply `'Size` to a type, the resulting value depends on the kind of type.

For scalar types, the attribute returns the *minimum* number of bits required to represent all the values of the type. Here's a diagram showing what the category "scalar types" includes:



Consider type **Boolean**, which has two possible values. One bit will suffice, and indeed the language standard requires `Boolean'Size` to be the value 1.

This meaning also applies to subtypes, which can constrain the number of values for a scalar type. Consider subtype **Natural**. That's a subtype defined by the language to be type **Integer** but with a range of `0 .. Integer'Last`. On a 32-bit machine we would expect **Integer** to be a native type, and thus 32-bits. On such a machine if we say `Integer'Size` we will indeed get 32. But if we say `Natural'Size` we will get 31, not 32, because only 31 bits are needed to represent that range on that machine.

The size of objects, on the other hand, cannot be just a matter of the possible values. Consider type **Boolean** again, where `Boolean'Size` is required to be 1. No compiler is likely to

allocate one bit to a **Boolean** variable, because typical machines don't support individually-addressable bits. Instead, addresses refer to storage elements, of a size indicated by the `Storage_Unit` constant. The compiler will allocate the smallest number of storage elements necessary, consistent with other considerations such as alignment. Therefore, for a machine that has `Storage_Unit` set to a value of eight, we can assume that a compiler for that machine will allocate an entire eight-bit storage element to a stand-alone **Boolean** variable. The other seven bits are simply not used by that variable. Moreover, those seven bits are not used by any other stand-alone object either, because access would be far less efficient, and such sharing would require some kind of locking to prevent tasks from interfering with each other when accessing those stand-alone objects. (Stand-alone objects are independently addressable; they wouldn't stand alone otherwise.)

By the same token (and still assuming a 32-bit machine), a compiler will allocate more than 31 bits to a variable of subtype `Natural` because there is no 31-bit addressable unit. The variable will get all 32-bits.

Note that we're talking about individual, stand-alone variables. Components of composite types, on the other hand, might indeed share bytes if the individual components don't require all the bits of their storage elements. You'd have to request that representation, though, with most implementations, because accessing the components at run-time would require more machine instructions. We'll go into the details of that later.

Let's talk further about sizes of types.

For record types, '`Size`' gives the minimum number of bits required to represent the whole composite value. But again, that's not necessarily the number of bits required for the objects' in-memory representation. The order of the components within the record can make a difference, as well as their alignments. The compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. As a result the overall size could be larger.

Note that Ada compilers are allowed to reorder the components; the order in memory might not match the order in the source code.

For example, consider this record type and its components:

```
type My_Int is range 1..10;
```

```
subtype S is Integer range 1..10;
```

```
type R is record
```

```
  M : My_Int;
```

```
  X : S;
```

```
  B : Boolean;
```

```
  C : Character;
```

```
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler

3	2	1	0
			M
X	X	X	X
		C	B

R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

In the figure, we see a record type with some components, and a sample layout for that record type assuming the compiler does not reorder the components. Observe that some bytes allocated to objects of type `R` are unused (the darkly shaded ones). In this case that's because the alignment of subtype `S` happens to be 4 on this machine. The component `X` of that subtype `S` cannot start at byte offset 1, or 2, or 3, because those addresses would not satisfy the alignment constraint of `S`. (We're assuming byte 0 is at a word-aligned address.) Therefore, `X` starts at the object's starting address plus 4. Components `B` and `C` are of types that have an alignment of 1, so they can start at any storage element. They immediately follow the bytes allocated to component `X`. Therefore, `R'Size` is 80, or 10 bytes. The three bytes following component `M` are simply not used.

But what about the two bytes following the last component `C`? They could be allocated to

stand-alone objects if they would fit. More likely, though, the compiler will allocate those two bytes to objects of type R, that is, 12 bytes instead of 10 are allocated. As a result, 96 bits are actually used in memory. The extra, unused 16 bits are "padding."

Why add unused padding? It simplifies the memory allocation of objects of type R. Suppose some array type has components of record type R. Assuming the first component is aligned properly, every following component will also be aligned properly, automatically, because the two padding bytes are considered parts of the components.

To make that work, the compiler takes the most stringent alignment of all the record type's components and uses that for the alignment of the overall record type. That way, any address that satisfies the record object's alignment will satisfy the components' alignment requirements. The alignment is component X, of subtype S, is 4. The other components have an alignment of 1, therefore R'Alignment is 4. An aligned address plus 12 will also be an aligned address.

This rounding up based on alignment is recommended behavior for the compiler, not a requirement, but is reasonable and typical among vendors. Although it can result in unused storage, that's the price paid for speed of access (or even correctness for machines that would fault on misaligned component accesses).

As you can see, alignment is a critical factor in the sizes of composite objects. If you care about the layout of the type you very likely need to care about the alignment of the components and overall record type.

Ada compilers are allowed to reorder the components of record types in order to minimize these gaps or satisfy the alignment requirements of the components. Some compilers do, some don't. Consider the type R again, this time with the first two components switched in the component declaration order:

```
type My_Int is range 1..10;
```

```
subtype S is Integer range 1..10;
```

```
type R is record
```

```
  X : S;
```

```
  M : My_Int;
```

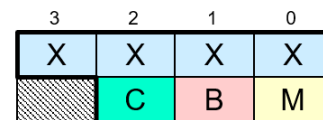
```
  B : Boolean;
```

```
  C : Character;
```

```
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler



R'Size will be 56 bits (7 bytes,
but all 8 will be allocated)

Now R'Size will report 56 bits instead of 80. The one trailing byte will still be padding, but only that one.

What about unbounded types, for example type **String**? Querying the 'Size in that case would provide an implementation-defined result. A somewhat silly thing to do, really, since the type — by definition — doesn't specify how many components are involved.

Usually, though, you don't want to query the size of a type. Most of the time what you want is the size of objects of the type. Going back to sending values over the wire, the code should query the size of the *parameter* holding the value to be sent. That will tell you how many bits are really needed.

One last point: GNAT, and now Ada 202x, define an attribute named `Object_Size`. It does just what the name suggests: what 'Size does when applied to objects rather than types. GNAT also defines another attribute, named `Value_Size`, that does what 'Size does when applied to types. The former is far more useful so Ada has standardized it.

75.5 Specifying Representation

Recall that we said **Boolean**'**Size** is required to be 1, and that stand-alone objects of type **Boolean** are very likely allocated some integral number of storage elements (e.g., bytes) in memory, typically one. What about arrays of Booleans? Suppose we have an array of 16 Boolean components. How big are objects of the array type? It depends on the machine. Continuing with our hypothetical (but typical) byte-addressable machine, for the sake of efficient access each component is almost certainly allocated an individual byte rather than a single bit, just like stand-alone objects. Consequently, our array of 16 Booleans will be reported by '**Size** to be 128 bits, i.e., 16 bytes. If you wanted a bit-mask, in which each Boolean component is allocated a single bit and the total array size is 16 bits, you'd have a problem. The compiler assumes you want speed of access rather than storage minimization, and normally that would be the right assumption.

Naturally there is a solution. Ada allows us to specify the representation characteristics of types, and thus objects of those types, including their bit-wise layouts. It also allows us to specify the representation of individual objects. You should understand, though, that the compiler is not required to do what you ask, because you might ask the impossible. For example, if you specify that the array of 16 Booleans is to be represented completely in 15 bits, what can the compiler do? Rejecting that specification is the only reasonable response. But if you specify something possible, the compiler must do what you ask, absent some compelling reason to the contrary.

With that in mind, let's examine setting the size for types.

So, how do we specify that we want our array of 16 Boolean components to be allocated one bit per component, for a total allocation of 16 bits? There are a couple of ways, one somewhat better than the other.

First, you can ask that the compiler "pack" the components into as small a number of bits as it can:

```
type Bits16 is array (0 .. 15) of Boolean with
  Pack;
```

That likely does what you want: Bits16'**Size** will probably be 16.

But realize that the Pack aspect (and corresponding pragma) is merely a request that the compiler do its best to minimize the number of bits allocated, not necessarily that it do exactly what you expected or required.

We could set the size of the entire array type:

```
type Bits16 is array (0 .. 15) of Boolean with
  Size => 16;
```

But the language standard says that a Size clause on array and record types should not affect the internal layout of their components. That's Implementation Advice, so not normative, but implementations are really expected to follow the advice, absent some compelling reason. That's what the Pack aspect, record representation clauses, and Component_Size clauses are for. (We'll talk about record representation clauses momentarily.) That said, at least one other vendor's compiler would have changed the size of the array type because of the Size clause, so GNAT defines a configuration pragma named Implicit_Packing that overrides the default behavior. With that pragma applied, the Size clause would compile and suffice to make the overall size be 16. That's a vendor-defined pragma though, so not portable.

Therefore, the best way to set the size for the array type is to set the size of the individual components, via the Component_Size aspect as the Implementation Advice indicates. That will say what we really want, rather than a "best effort" request for the compiler, and is portable:

```
type Bits16 is array (0 .. 15) of Boolean with
  Component_Size => 1;
```

With this approach the compiler must either use the specified size for each component or refuse to compile the code. If it compiles, objects of the array type will be 16 bits total (plus any padding bits required to make objects have a size that is a multiple of `Storage_Unit`, typically zero on modern machines).

Now that we have a bit-mask array type, let's put it to use.

Let's say that you have an object that is represented as a simple signed integer because, for most usage, that's the appropriate representation. Sometimes, though, let's say you need to access individual bits of the object instead of the whole numeric value. Signed integer types don't provide bit-level access. In Ada we'd say that the "view" presented by the object's type doesn't include bit-oriented operations. Therefore, we need to add a view to the object that does provide them. A different view will require an additional type for the same object.

Applying different types, and thus their operations, to the same object is known as [type punning](https://en.wikipedia.org/wiki/Type_punning)⁴²³ in computer programming. Realize that doing so circumvents the static strong typing we harness to protect us from ourselves and from others. Use it with care! (For example, limit the compile-time visibility to such code.)

One way to add a view is to express an "overlay," in which an object of one type is placed at the same memory location as a distinct object of a different type, thus "overlying" one object over the other in memory. The different types present different views, therefore different operations available for the shared memory cells. Our hypothetical example uses two views, but you can overlay as many different views as needed. (That said, requiring a large number of different views of the same object would be suspect.)

There are other ways in Ada to apply different views, some more flexible than others, but an overlay is a simple one that will often suffice.

Here is an implementation of the overlay approach, using our bit-mask array type:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;
```

```
X : Integer;
Y : Bits32 with Address => X'Address;
```

We can query the addresses of objects, and other things too, but objects, especially variables, are the most common case. In the above, we say `X'Address` to query the starting address of object `X`. With that information we know what address to specify for our bit-mask overlay object `Y`. Now `X` and `Y` are aliases for the same memory cells, and therefore we can manipulate and query that memory as either a signed integer or as an array of bits. Reading or updating individual array components accesses the individual bits of the overlaid object.

Instead of the `Bits32` array type, we could have specified a modular type for the overlay `Y` to get a view providing bit-oriented operations. Overlaying such an array was a common idiom prior to the introduction of modular "unsigned" types in Ada, and remains useful for accessing individual bits. In other words, using a modular type for `Y`, you could indeed access an individual bit by passing a mask value to the `and` operator defined in any modular type's view. Using a bit array representation lets the compiler do that work for you, in the generated code. The source code will be both easier to read and more explicit about what it is doing when using the bit array overlay.

One final issue remains: in our specific overlay example the compiler would likely generate code that works. But strictly speaking it might not.

⁴²³ https://en.wikipedia.org/wiki/Type_punning

The Ada language rules say that for such an overlaid object — Y in the example above — the compiler should not perform optimizations regarding Y that it would otherwise apply in the absence of aliases. That's necessary, functionally, but may imply degraded performance regarding Y, so keep it in mind. Aliasing precludes some desirable optimizations.

But what about X in the example above? We're querying that object's address, not specifying it, so the RM rule precluding optimizations doesn't apply to X. That can be problematic.

The compiler might very well place X in a register, for example, for the sake of the significant performance increase (another way of being friendly). But in that case `System.Null_Address` will be returned by the `X'Address` query and, consequently, the declaration for Y will not result in the desired overlaying.

Therefore, we should mark X as explicitly **aliased** to ensure that `X'Address` is well-defined:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

The only difference in the version above is the addition of **aliased** in the declaration of X. Now we can be certain that the optimizer will not represent X in some way incompatible with the idiom, and `X'Address` will be well-defined.

In our example X and Y are clearly declared in the same compilation unit. Most compilers will be friendly in this scenario, representing X in such a way that querying the address will return a non-null address value even if **aliased** is not applied. Indeed, **aliased** is relatively new to Ada, and earlier compilers typically emitted code that would handle the overlay as intended.

But suppose, instead of being declared in the same declarative part, that X was declared in some other compilation unit. Let's say it is in the visible part of a package declaration. (Assume X is visible to clients for some good reason.) That package declaration can be, and usually will be, compiled independently of clients, with the result that X might be represented in some way that cannot support querying the address meaningfully.

Therefore, the declaration of X in the package spec should be marked as aliased, explicitly:

```
package P is
  X : aliased Integer;
end P;
```

Then, in the client code declaring the overlay, we only declare Y, assuming a with-clause for P:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

Y : Bits32 with Address => P.X'Address;
```

All well and good, but how did the developer of the package know that some other unit, a client of the package, would query the address of X, such that it needed to be marked as aliased? Indeed, the package developer might not know. Yet the programmer is responsible for ensuring a valid and appropriate **Address** value is used in the declaration of Y. Execution is erroneous otherwise, so we can't say what would happen in that case. Maybe an exception is raised or a machine trap, maybe not.

Worse, the switches that were applied when compiling the spec for package P can make a difference: `P.X` might not be placed in a register unless the optimizer is enabled. Hence the client code using Y might work as expected when built for debugging, with the optimizer disabled, and then not do so when re-built for the final release. You'd probably have to solve this issue by debugging the application.

On a related note, you may be asking yourself how to know that type **Integer** is 32 bits wide, so that we know what size array to use for the bit-mask. The answer is that you just have to know the target well when doing low-level programming. The hardware becomes much more visible, as we mentioned.

That said, you could at least verify the assumption:

```
pragma Compile_Time_Error (Integer'Object_Size /= 32,  
                           "Integers expected to be 32 bits");  
X : aliased Integer;  
Y : Bits32 with Address => X'Address;
```

That's a vendor-defined pragma so this is not fully portable. It isn't an unusual pragma, though, so at least you can probably get the same functionality even if the pragma name varies.

Overlays aren't always structured like our example above, i.e., with two objects declared at the same time. We might apply a different type to the same memory locations at different times. Here's an example from the ADL to illustrate the idea. We'll elaborate on this example later, in another section.

First, a package declaration, with two functions that provide a device-specific unique identifier located in shared memory. Each function provides the same Id value in a distinct format. One format is a string of 12 characters, the other is a sequence of three 32-bit values. Hence both representations are the same size.

```
package STM32.Device_Id is  
  
    subtype Device_Id_Image is String (1 .. 12);  
  
    function Unique_Id return Device_Id_Image;  
  
    type Device_Id_Tuple is array (1 .. 3) of UInt32  
        with Component_Size => 32;  
  
    function Unique_Id return Device_Id_Tuple;  
  
end STM32.Device_Id;
```

In the package body we implement the functions as two ways to access the same shared memory, specified by ID_Address:

```
with System;  
  
package body STM32.Device_Id is  
  
    ID_Address : constant System.Address := System.To_Address (16#1FFF_7A10#);  
  
    function Unique_Id return Device_Id_Image is  
        Result : Device_Id_Image with Address => ID_Address, Import;  
    begin  
        return Result;  
    end Unique_Id;  
  
    function Unique_Id return Device_Id_Tuple is  
        Result : Device_Id_Tuple with Address => ID_Address, Import;  
    begin  
        return Result;  
    end Unique_Id;  
  
end STM32.Device_Id;
```

System'To_Address is just a convenient way to convert a numeric value into an **Address**

value. The primary benefit is that the call is a static expression, but we can ignore that here. Using `Import` is a good idea to ensure that the Ada code does no initialization of the object, since the value is coming from the hardware via the shared memory. Doing so may not be necessary, depending on the type used, but is a good habit to develop.

The point of this example is that we have one object declaration per function, of a type corresponding to the intended function result type. Because each function places their local object at the same address, they are still overlaying the shared memory.

Now let's return, momentarily, to setting the size of entities, but now let's focus on setting the size of objects.

We've said that the size of an object is not necessarily the same as the size of the object's type. The object size won't be smaller, but it could be larger. Why? For a stand-alone object or a parameter, most implementations will round the size up to a storage element boundary, or more, so the object size might be greater than that of the type. Think back to **Boolean**, where `Size` is required to be 1, but stand-alone objects are probably allocated 8 bits, i.e., an entire storage element (on our hypothetical byte-addressed machine).

Likewise, recall that numeric type declarations are mapped to underlying hardware numeric types. These underlying numeric types provide at least the capabilities we request with our type declarations, e.g., the range or number of digits, perhaps more. But the mapped numeric hardware type cannot provide less than requested. If there is no underlying hardware type with at least our requested capabilities, our declarations won't compile. That mapping means that specifying the size of a numeric type doesn't necessarily affect the size of objects of the type. That numeric hardware type is the size that it is, and is fixed by the hardware.

For example, let's say we have this declaration:

```
type Device_Register is range 0 .. 2**5 - 1 with Size => 5;
```

That will compile successfully, because there will be a signed integer hardware type with at least that range. (Not necessarily, legally speaking, but realistically speaking, there will be such a hardware type.) Indeed, it may be an 8-bit signed integer, in which case `Device_Register'Size` will give us 5, but objects of the type will have a size of 8, unavoidably, even though we set `Size` to 5.

The difference between the type and object sizes can lead to potentially problematic code:

```
type Device_Register is range 0 .. 2**8 - 1 with Size => 8;
```

```
My_Device : Device_Register
  with Address => To_Address (...);
```

The code compiles successfully, and tries to map a byte to a hardware device that is physically connected to one storage element in the processor memory space. The actual address is elided as it is not important here.

That code might work too, but it might not. We might think that `My_Device'Size` is 8, and that `My_Device'Address` points at an 8-bit location. However, this isn't necessarily so, as we saw with the supposedly 5-bit example earlier. Maybe the smallest signed integer the hardware has is 16-bits wide. The code would compile because a 16-bit signed numeric type can certainly handle the 8-bit range requested. `My_Device'Size` would be then 16, and because `'Address` gives us the *starting* storage element, `My_Device'Address` might designate the high-order byte of the overall 16-bit object. When the compiler reads the two bytes for `My_Device` what will happen? One of the bytes will be the data presented by the hardware device mapped to the memory. The other byte will contain undefined junk, whatever happens to be in the memory cell at the time. We might have to debug the code a long time to identify that as the problem. More likely we'll conclude we have a failed device.

The correct way to write the code is to specify the size of the object instead of the type:


```
type Device_Register is range 0 .. 2**8 - 1;
```

```
My_Device : Device_Register with  
  Size => 8,  
  Address => To_Address (...);
```

If the compiler cannot support stand-alone 8-bit objects for the type, the code won't compile.

Alternatively, we could change the earlier Size clause on the type to apply Object_Size instead:

```
type Device_Register is range 0 .. 2**8 - 1 with Object_Size => 8;
```

```
My_Device : Device_Register with  
  Address => To_Address (...);
```

The choice between the two approaches comes down to personal preference, at least if only a small number of stand-alone objects of the type are going to be declared. With either approach, if the implementation cannot support 8-bit stand-alone objects, we find out that there is a problem at compile-time. That's always cheaper than debugging.

You might conclude that setting the Size for a type serves no purpose. That's not an unreasonable conclusion, given what you've seen, but in fact there are reasons to do so. However, there are only a few specific cases so we will save the reasons for the discussions of the specific cases.

There is one general case, though, for setting the 'Size of a type. Specifically, you may want to specify the size that you think is the minimum possible, and you want the compiler to confirm that belief. This would be one of the so-called "confirming" representation clauses, in which the representation detail is what the compiler would have chosen anyway, absent the specification. You're not actually changing anything, you're just getting confirmation via Size whether or not the compiler accepts the clause. Suppose, for example, that you have an enumeration type with 256 values. For enumeration types, the compiler allocates the smallest number of bits required to represent all the values, rounded up to the nearest storage element. (It's not like C, where enums are just named int values.) For 256 values, an eight-bit byte would suffice, so setting the size to 8 would be confirming. But suppose we actually had 257 enumerals, accidentally? Our size clause set to 8 would not compile, and we'd be told that something is amiss.

However, note that if your supposedly "confirming" size clause actually specifies a size larger than what the compiler would have chosen, you won't know, because the compiler will silently accept sizes larger than necessary. It just won't accept sizes that are too small.

There are other confirming representation clauses as well. Thinking again of enumeration types, the underlying numeric values are integers, starting with zero and consecutively increasing from there up to N-1, where N is the total number of enumerals values.

For example:

```
type Commands is (Off, On);
```

```
for Commands use (Off => 0, On => 1);
```

As a result, Off is encoded as 0 and On as 1. That specific underlying encoding is guaranteed by the language, as of Ada 95, so this is just a confirming representation clause nowadays. But it was not guaranteed in the original version of the language, so if you wanted to be sure of the encoding values you would have specified the above. It wasn't necessarily confirming before Ada 95, in other words.

But let's also say that the underlying numeric values are not what you want because you're interacting with some device and the commands are encoded with values other than 0 and

1. Maybe you want to use an enumeration type because you want to specify all the possible values actually used by clients. If you just used some numeric type instead and made up constants for `On` and `Off`, there's nothing to keep clients from using other numeric values in place of the two constants (absent some comparatively heavy code to prevent that from happening). Better to use the compiler to make that impossible in the first place, rather than debug the code to find the incorrect values used. Therefore, we could specify different encodings:

```
for Commands use (Off => 2, On => 4);
```

Now the compiler will use those encoding values instead of 0 and 1, transparently to client code.

The encoding values specified must maintain the relative ordering, otherwise the relational operators won't work correctly. For example, for type `Commands` above, `Off` is less than `On`, so the specified encoding value for `Off` must be less than that of `On`.

Note that the values given in the example no longer increase consecutively, i.e., there's a gap. That gap is OK, in itself. As long as we use the two enumerals the same way we'd use named constants, all is well. Otherwise, there is both a storage issue and a performance issue possible. Let's say that we use that enumeration type as the index for an array type. Perfectly legal, but how much storage is allocated to objects of this array type? Enough for exactly two components? Four, with two unused? The answer depends on the compiler, and is therefore not portable. The bigger the gaps, the bigger the overall storage difference possible. Likewise, imagine we have a for-loop iterating over the index values of one of these array objects. The for-loop parameter cannot be coded by the compiler to start at 0, clearly, because there is no index (enumeration) value corresponding to 0. Similarly, to get the next index, the compiler cannot have the code simply increment the current value. Working around that takes some extra code, and takes some extra time that would not be required if we did not have the gaps.

The performance degradation can be significant compared to the usual code generated for a for-loop. Some coding guidelines say that you shouldn't use an enumeration representation clause for this reason, with or without gaps. Now that Ada has type predicates we could limit the values used by clients for a numeric type, so an enumeration type is not the only way to get a restricted set of named, encoded values.

```
type Commands is new Integer with
  Static_Predicate => Commands in 2 | 4;

On    : constant Commands := 2;
Off   : constant Commands := 4;
```

The storage and performance issues bring us back to confirming clauses. We want the compiler to recognize them as such, so that it can generate the usual code, thereby avoiding the unnecessary portability and performance issues. Why would we have such a confirming clause now? It might be left over from the original version of the language, written before the Ada 95 change. Some projects have lifetimes of several decades, after all, and changing the code can be expensive (certified code, for example). Whether the compiler does recognize confirming clauses is a feature of the compiler implementation. We can expect a mature compiler to do so, but there's no guarantee.

Now let's turn to what is arguably the most common representation specification, that of record type layouts.

Recall from the discussion above that Ada compilers are allowed to reorder record components in physical memory. In other words, the textual order in the source code is not necessarily the physical order in memory. That's different from, say, C, where what you write is what you get, and you better know what you're doing. On some targets a misaligned `struct` component access will perform very poorly, or even trap and halt, but that's not the C compiler's fault. In Ada you'd have to explicitly specify the problematic layout. Otherwise, if compilation is successful, the Ada compiler must find a representation that will

work, either by reordering the components or by some other means. Otherwise it won't compile.

GNAT did not reorder components until relatively recently but does now, at least for the more egregious performance cases. It does this reordering silently, too, although there is a switch to have it warn you when it does. To prevent reordering, GNAT defines a pragma named `No_Component_Reorder` that does what the name suggests. You can apply it to individual record types, or globally, as a configuration pragma. But of course because the pragma is vendor defined it is not portable.

Therefore, if you care about the record components' layout in memory, the best approach is to specify the layout explicitly. For example, perhaps you are passing data to code written in C. In that case, you need the component order in memory to match the order given in the corresponding C struct declaration. That order in memory is not necessarily guaranteed from the order in the Ada source code. The Ada compiler is allowed to choose the representation unless you specify it, and it might choose a different layout from the one given. (Ordinarily, letting the compiler choose the layout is the most desirable approach, but in this case we have an external layout requirement.)

Fortunately, specifying a record type's layout is straightforward. The record layout specification consists of the storage places for some or all components, specified with a record representation clause. This clause specifies the order, position, and size of components (including discriminants, if any).

The approach is to first define the record type, as usual, using any component order you like — you're about to specify the physical layout explicitly, in the next step.

Let's reuse that record type from the earlier discussion:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

The resulting layout might be like so, assuming the compiler doesn't reorder the components:

```
type My_Int is range 1..10;
```

```
subtype S is Integer range 1..10;
```

```
type R is record
```

```
  M : My_Int;
```

```
  X : S;
```

```
  B : Boolean;
```

```
  C : Character;
```

```
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler

3	2	1	0
			M
X	X	X	X
		C	B

R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

As a result, R'Size will be 80 bits (10 bytes), but those last two bytes will be allocated to objects, for an Object_Size of 96 bits (12 bytes). We'll change that with an explicit layout specification.

Having declared the record type, the second step consists of defining the corresponding record representation clause giving the components' layout. The clause uses syntax that somewhat mirrors that of a record type declaration. The components' names appear, as

in a record type declaration. But now, we don't repeat the components' types, instead we give their relative positions within the record, in terms of a relative offset that starts at zero. We also specify the bits we want them to occupy within the storage elements starting at that offset.

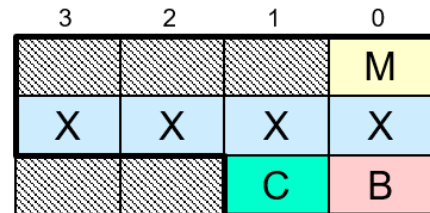
```
for R use record
  X at 0 range 0 .. 31; -- note the order swap,
  M at 4 range 0 .. 7;  -- with this component
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```

Now we'll get the optimized order, and we'll always get that order, or the layout specification won't compile in the first place. In the following diagram, both layouts, the default, and the one resulting from the record representation clause, are depicted for comparison:

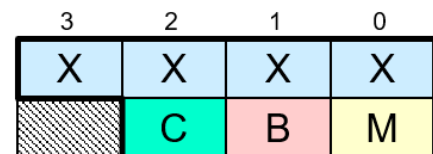
```
type My_Int is range 1..10;
```

```
subtype S is Integer range 1..10;
```

```
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```



```
for R use record
  X at 0 range 0 .. 31;
  M at 4 range 0 .. 7;
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```



R'Size will be 56 bits (7 bytes), but that last padding byte will also be allocated to objects, so the Object_Size will be 64 bits (8 bytes).

Notice how we gave each component an offset, after the reserved word **at**. These offsets are in terms of storage elements, and specify their positions within the record object as a whole. They are relative to the beginning of the memory allocated to the record object so they are numbered starting at zero. We want the X component to be the very first component in the allocated memory so the offset for that one is zero. The M component, in comparison, starts at an offset of 4 because we are allocating 4 bytes to the prior component X: bytes 0 through 3 specifically. M just occupies one storage element so the next component, B, starts at offset 5. Likewise, component C starts at offset 6.

Note that there is no requirement for the components in the record representation clause to be in any particular textual order. The offsets alone specify the components' order in memory. A good style, though, is to order the components in the representation clause so that their textual order corresponds to their order in memory. Doing so facilitates our verifying that the layout is correct because the offsets will be increasing as we read the

specification.

An individual component may occupy part of a single storage element, all of a single storage element, multiple contiguous storage elements, or a combination of those (i.e., some number of whole storage elements but also part of another). The bit "range" specifies this bit-specific layout, per component, by specifying the first and last bits occupied. The X component occupies 4 complete 8-bit storage elements, so the bit range is 0 through 31, for a total of 32 bits. All the other components each occupy an entire single storage element so their bit ranges are 0 through 7, for a total of 8 bits.

The text specifying the offset and bit range is known as a "component_clause" in the syntax productions. Not all components need be specified by component_clauses, but (not surprisingly) at most one clause is allowed per component. Really none are required but it would be strange not to have some. Typically, all the components are given positions. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation.

Components not otherwise given an explicit placement are given positions chosen by the compiler. We don't say that they "follow" those explicitly positioned because there's no requirement that the explicit positions start at offset 0, although it would be unusual not to start there.

Placements must not make components overlap, except for components of variant parts, a topic covered elsewhere. You can also specify the placement of implementation-defined components, as long as you have a name to refer to them. (In addition to the components listed in the source code, the implementation can add components to help implement what you wrote explicitly.) Such names are always attribute references but the specific attributes, if any, are implementation-defined. It would be a mistake for the compiler to define such implicit components without giving you a way to refer to them. Otherwise they might go exactly where you want some other component to be placed, or overlap that place.

The positions (offsets) and the bit numbers must be static, informally meaning that they are known at compile-time. They don't have to be numeric literals, though. Numeric constants would work, but literals are the most common by far.

Note that the language does not limit support for component clauses to specific component types. They need not be one of the integer types, in particular. For example, a position can be given for components that are themselves record types, or array types. Even task types are allowed as far as the language goes, although the implementation might require a specific representation, such as the component taking no bits whatsoever (0 .. -1). There are restrictions that keep things sane, for example rules about how a component name can be used within the overall record layout construct, but not restrictions on the types allowed for individual components. For example, here is a record layout containing a **String** component, arbitrarily set to contain 11 characters:

```
type R is record
  S : String (1 .. 11);
  B : Boolean;
end record;

for R use record
  S at 0 range 0 .. 87;
  B at 11 range 0 .. 7;
end record;
```

Component S is to be the first component in memory in this example, hence the position offset is 0, for the first byte of S. Next, S is 11 characters long, or 88 bits, so the bit range is 0 .. 87. That's 11 bytes of course, so S occupies storage elements 0 .. 10. Therefore, the next component position must be at least 11, unless there is to be a gap, in which case it would be greater than 11. We'll place B immediately after the last character of S, so B is at storage element offset 11 and occupying all that one byte's bits.

We'll have more to say about record type layouts but first we need to talk about alignment.

Modern target architectures are comparatively strict about the address alignments for some of their types. If the alignment is off, an access to the memory for objects of the type can have highly undesirable consequences. Some targets will experience seriously degraded performance. On others, the target will halt altogether. As you can see, getting the alignment correct is a low-level, but vital, part of correct code on these machines.

Normally the compiler does this work for us, choosing an alignment that is both possible for the target and also optimal for speed of access. You can, however, override the compiler's alignment choice using an attribute definition clause or the `Alignment` aspect. You can do so on types other than record types, but specifying it on record types is typical. Here's our example record type with the alignment specified via the aspect:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record with
  Alignment => 1;
```

Alignment values are in terms of storage elements. The effect of the aspect or attribute clause is to ensure that the starting address of the memory allocated to objects of the type will be a multiple of the specified value.

In fact, whenever we specify a record type layout we really should also specify the record type's alignment, even though doing so is optional. Why? The alignment makes a difference in the overall record object's size. We've seen that already, with the padding bytes: the compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. The alignment also affects the size allocated to the record type even when the components are already aligned. As a result the overall size could be larger than we want for the sake of space. Additionally, when we pass such objects to code written in other languages, we want to ensure that the starting address of these objects is aligned as the external code expects. The compiler might not choose that required alignment by default.

Specifying alignment for record types is so useful that in the first version of Ada there was no syntax to specify alignment for anything other than record types (via the obsolete `at mod` clause on record representation clauses).

For that reason GNAT provides a pragma named `Optimize_Alignment`. This is a configuration pragma that affects the compiler's choice of default alignments where no alignment is explicitly specified. There is a time/space trade-off in the selection of these values, as we've seen. The normal choice tries to balance these two characteristics, but with an argument to the pragma you can give more weight to one or the other. The best approach is to specify the alignments explicitly, per type, for those that require specific alignment values. The pragma has the nice property of giving general guidance to the compiler for what should be done for the other types and objects not explicitly specified.

Now let's look into the details. We'll use a case study for this purpose, including specifying sizes as well as alignments.

The code for the case study is as follows. It uses `Size` clauses to specify the Sizes, instead of the `Size` aspect, just to emphasize that the `Size` clause approach is not obsolete.

```
package Some_Types is

  type Temperature is range -275 .. 1_000;
```

(continues on next page)

(continued from previous page)

```
type Identity is range 1 .. 127;

type Info is record
  T : Temperature;
  Id : Identity;
end record;

for Info use record
  T at 0 range 0 .. 15;
  Id at 2 range 0 .. 7;
end record;

for Info'Size use 24;

type List is array (1 .. 3) of Info;
for List'Size use 24 * 3;

end Some_Types;
```

When we compile this, the compiler will complain that the size for `List` is too small, i.e., that the minimum allowed is 96 bits instead of the 72 we specified. We specified `24 * 3` because we said the record size should be 24 bits, and we want our array to contain 3 record components of that size, so 72 seems right.

What's wrong? As we've shown earlier, specifying the record type size doesn't necessarily mean that objects (in this case array components) are that size. The object size could be bigger than we specified for the type. In this case, the compiler says we need 96 total bits for the array type, meaning that each of the 3 array components is 32 bits wide instead of 24.

Why is it 32 bits? Because the alignment for `Info` is 2 (on this machine). The record alignment is a multiple of the largest alignment of the enclosed components. The alignment for type `Temperature` (2), is larger than the alignment for type `Identity` (1), therefore the alignment for the whole record type is 2. We need to go from that number of storage elements to a number of bits for the size.

Here's where it gets subtle. The alignment is in terms of storage elements. Each storage element is of a size in bits given by `System.Storage_Unit`. We've said that on our hypothetical machine `Storage_Unit` is 8, so storage elements are 8 bits wide on this machine. Bytes, in other words. Therefore, to get the required size in bits, we have to find a multiple of the two 8-bit bytes (specified by the alignment) that has at least the number of bits we gave in the `Size` clause. Two bytes only provides 16 bits, so that's not big enough, we need at least 24 bits. The next multiple of 2 bytes is 4 bytes, providing 32 bits, which is indeed larger than 24. Therefore, the overall size of the record type, consistent with the alignment, is 4 bytes, or 32 bits. That's why the compiler says each array component is 32 bits wide.

But for our example let's say that we really want to use only 72 total bits for the array type (and that we want three array components). That's the size we specified, after all. So how do we get the record type to be 24 bits instead of 32? Yes, you guessed it, we change the alignment for the record type. If we change it from 2 to 1, the size of 24 bits will work. Adding this `Alignment` clause line will do that:

```
for Info'Alignment use 1;
```

An alignment of 1 means that any address will work, assuming that addresses refer to entire storage elements. (An alignment of 0 would mean that the address need not start on a storage element boundary, but we know of no such machines.)

We can even entirely replace the `Size` clause with the `Alignment` clause, because the `Size` clause specifying 24 bits is just confirming: it's the value that `'Size` would return anyway.

The problem is the object size.

Now, you may be wondering why an alignment of 1 would work, given that the alignment of the Temperature component is 2. Wouldn't it slow down the code, or even trap? Well, maybe. It depends on the machine. If it doesn't work we would just have to use 32 bits for the record type, with the original alignment of 2, for a larger total array size. Of course, if the compiler recognizes that a representation cannot be supported it must reject the code, but the compiler might not recognize the problem.

We said earlier that there are only a small number of reasons to specify 'Size for a type. We can mention one of them now. Setting 'Size can be useful to give the minimum number of bits to use for a component of a packed composite type, that is, within either a record type or an array type that is explicitly packed via the aspect or pragma Pack. It says that the compiler, when giving its best effort, shouldn't compress components of the type any smaller than the number of bits specified. No, it isn't earth-shattering, but other uses are more valuable, to be discussed soon.

One thing we will leave unaddressed (pun intended) is the question of bit ordering and byte ordering within our record layouts. In other words, the "endian-ness". That's a subject beyond the scope of this course. Suffice it to say that GNAT provides a way to specify record layouts that are independent of the endian-ness of the machine, within some implementation-oriented limits. That's obviously useful when the code might be compiled for a different ISA in the future. On the other hand, if your code is specifically for a single ISA, e.g. Arm, even if different boards and hardware vendors are involved, there's no need to be independent of the endian-ness. It will always be the same in that case. (Those are "famous last words" though.) For an overview of the GNAT facility, an attribute named attribute Scalar_Storage_Order see <https://www.adacore.com/papers/lady-ada-mediates-peace-treaty-in-endianness-war>.

Although specifying record type layouts and alignments are perhaps the most common representation characteristics expressed, there are a couple of other useful cases. Both involve storage allocation.

One useful scenario concerns tasking. We can specify the number of storage elements reserved for the execution of a task object, or all objects of a task type. You use the Storage_Size aspect to do so:

```
task Servo with
  Storage_Size => 1 * 1024,
  ...
```

Or the corresponding pragma:

```
task Servo is
  pragma Storage_Size (1 * 1024);
end Servo;
```

The aspect seems textually cleaner and lighter unless you have task entries to declare as well. In that case the line for the pragma wouldn't add all that much. That's a matter of personal aesthetics anyway.

The specified number of storage elements includes the size of the task's stack (GNAT does have one, per task). The language does not specify whether or not it includes other storage associated with the task used for implementing and managing the task execution. With GNAT, the extent of the primary stack size is the value returned, ignoring any other storage used internally in the run-time library for managing the task.

The GNAT run-time library allocates a default stack amount to each task, with different defaults depending on the underlying O.S., or lack thereof, and the target. You need to read the documentation to find the actual amount, or, with GNAT, read the code.

You would need to specify this amount in order to either increase or decrease the allocated storage. If the task won't run properly, perhaps crashing at strange and seemingly random

places, there's a decent chance it is running out of stack space. That might also be the reason if you have a really deep series of subprogram calls that fails. The correction is to increase the allocation, as shown above. How much? Depends on the application code. The quick-and-dirty approach is to iteratively increase the allocation until the task runs properly. Then, reverse the approach until it starts to fail again. Add a little back until it runs, and leave it there. We'll mention a much better approach momentarily (**GNATstack**).

Even if the task doesn't seem to run out of task stack, you might want to reduce it anyway, to the extent possible, because the total amount of storage on your target might be limited. Some of the GNAT bare-metal embedded targets have very small amounts of memory available, so much so that the default task stack allocations would exhaust the memory available quickly. That's what the example above does: empirical data showed that the Servo task could run with just 1K bytes allocated, so we reduced it from the default accordingly. (We specified the size with that expression for the sake of readability, relative to using literals directly.)

Notice we said "empirical data" above. How do we know that we exercised the task's thread of control exhaustively, such that the arrived-at allocation value covers the worst case? We don't, not with certainty. If we really must know the allocation will suffice for all cases, say because this is a high-integrity application, we would use **GNATstack**. GNATstack is an offline tool that exploits data generated by the compiler to compute worst-case stack requirements per subprogram and per task. As a static analysis tool, its computation is based on information known at compile time. It does not rely on empirical run-time information.

The other useful scenario for allocating storage concerns access types, specifically access types whose values designate objects, as opposed to designating subprograms. (Remember, objects are either variables or constants.) There is no notion of dynamically allocating procedures and functions in Ada so access-to-subprogram types are not relevant here. But objects can be of protected types (or task types), and protected objects can "contain" entries and protected subprograms, so there's a lot of expressive power available. You just don't dynamically allocate procedures or functions as such.

First, a little background on access types, to supplement what we said earlier.

By default, the implementation chooses a standard storage pool for each named access-to-object type. The storage allocated by an allocator (i.e., **new**) for such a type comes from the associated pool.

Several access types can share the same pool. By default, the implementation might choose to have a single global storage pool, used by all such access types. This global pool might consist merely of calls to operating system routines (e.g., `malloc`), or it might be a vendor-defined pool instead. Alternatively, the implementation might choose to create a new pool for each access-to-object type, reclaiming the pool's memory when the access type goes out of scope (if ever). Other schemes are possible.

Finally, users may define new pool types, and may override the choice of pool for an access-to-object type by specifying `Storage_Pool` for the type. In this case, allocation (via **new**) takes memory from the user-defined pool and deallocation puts it back into that pool, transparently.

With that said, here's how to specify the storage to be used for an access-to-object type. There are two ways to do it.

If you specify `Storage_Pool` for an access type, you indicate a specific pool object to be used (user-defined or vendor-defined). The pool object determines how much storage is available for allocation via **new** for that access type.

Alternatively, you can specify `Storage_Size` for the access type. In this case, an implementation-defined pool is used for the access type, and the storage available is at least the amount requested, maybe more (it might round up to some advantageous block size, for example). If the implementation cannot satisfy the request, `Storage_Error` is raised.

It should be clear that the two alternatives are mutually exclusive. Therefore the compiler will not allow you to specify both.

Each alternative has advantages. If your only concern is the total number of allocations possible, use `Storage_Size` and let the implementation do the rest. However, maybe you also care about the behavior of the allocation and deallocation routines themselves, beyond just providing and reclaiming the storage. In that case, use `Storage_Pool` and specify a pool object of the appropriate type. For example, you (or the vendor, or someone else) might create a pool type in which the allocation routine performs in constant time, because you want to do **new** in a real-time application where predictability is essential.

Lastly, an idiom: when using `Storage_Size` you may want to specify a value of zero. That means you intend to do no allocations whatsoever, and want the compiler to reject the code if you try. Why would you want an access type that doesn't allow dynamically allocating objects? It isn't as unreasonable as it might sound. If you plan to use the access type strictly with aliased objects, never doing any allocations, you can have the compiler enforce your intent. There are application domains that prohibit dynamic allocations due to the difficulties in analyzing their behavior, including issues of fragmentation and exhaustion. Access types themselves are allowed in these domains. You'd simply use them to designate aliased objects alone. In addition, in this usage scenario, if the implementation associates an actual pool with each access type, the pool's storage would be wasted since you never intend to allocate any storage from it. Specifying a size of 0 tells the implementation not to waste that storage.

Before we end this section, there is a GNAT compiler switch you should know about. The `-gnatR?` switch instructs the compiler to list the representation details for the types, objects and subprograms in the compiled file(s). Both implementation-defined and user-defined representation details are presented. The '?' is just a placeholder and can be one of the following characters:

```
[0|1|2|3|4][e][j][m][s]
```

Increasing numeric values provide increasing amounts of information. The default is '1' and usually will suffice. See the GNAT User's Guide for Native Platforms for the details of the switch in [section 4.3.15 Debugging Control](#)⁴²⁴.

You'll have to scroll down some to find that specific switch but it is worth finding and remembering. When you cannot understand what the compiler is telling you about the representation of something, this switch is your best friend.

75.6 Unchecked Programming

Ada is designed to be a reliable language by default, based as it is on static strong typing and high-level semantics. Many of the pitfalls that a developer must keep in the back of their mind with other languages do not apply in Ada, and are typically impossible. That protection extends to low-level programming as well, e.g., the Separation Principle. Nevertheless, low-level programming occasionally does require mechanisms that allow us to go beyond the safety net provided by the type rules and high-level language constructs.

One such mechanism (unchecked conversion) provides a way to circumvent the type system, a system otherwise firmly enforced by the compiler on our behalf. Note that by "circumventing the type system" we do not include so-called "checked" conversions. These conversions have meaningful semantics, and are, therefore, allowed by the language using a specific syntax. This conversion syntax is known as "functional" syntax because it looks like a function call, except that the "function" name is a type name, and the parameter is the object or value being converted to that type. These conversions are said to be "checked" because only specific kinds of types are allowed, and the compiler checks that such conversions are indeed between these allowed types.

⁴²⁴ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#debugging-control

Instead, this section discusses "unchecked" programming, so-called because the compiler does not check for meaningful semantics. There are multiple mechanisms for unchecked programming in Ada: in addition to circumventing the type system, we can also deallocate a previously-allocated object, and can create an access value without the usual checks. In all cases the responsibility for correct meaning and behavior rests on the developer. Very few, if any, checks are done by the compiler. If we convert a value to another type that generally makes no sense, for example a task object converted to a record type, we are on our own. If we deallocate an allocated object more than once, it is our fault and Bad Things inevitably result.

Likened to "escape hatches," the facilities for unchecked programming are explicit in Ada. Their use is very clear in the source code, and is relatively heavy: each mechanism is provided by the language in the form of a generic library subprogram that must be specified in a context clause ("with-clause") at the top of the file, and then instantiated prior to use, like any generic. For an introduction to generic units in Ada, see that section in the introductory Ada course: [Introduction to Ada](#) (page 125)

You should understand that the explicitly unchecked facilities in Ada are no more unsafe than the implicitly unchecked facilities in other languages. There's no safety-oriented reason to "drop down" to C, for example, to do low-level programming. For that matter, the low-level programming facilities in Ada are at least as powerful as those in other languages, and probably more so.

We will explore unchecked storage deallocation in a separate book so let's focus on unchecked type conversions.

Unchecked type conversions are achieved by instantiating this language-defined generic library function, a "child" of the root package named "Ada":

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target
  with Pure, Nonblocking, Convention => Intrinsic;
```

The function, once instantiated and eventually invoked, returns the caller's value passed to S (of type Source) as if it is a value of type Target. That value can then be used in any way consistent with the Target type.

The two generic parameters, Source and Target, are defined in a manner that makes them very permissive in terms of the types they will accept when instantiated. To understand how, you need to understand a little bit of Ada's terminology and design for generic unit parameters. (If you are already familiar with generic formal types and how they are matched, feel free to skip this material.)

First, the terminology. The type parameters defined by a generic unit are known as "generic formal types," or "generic formals" for short. Types Source and Target are the generic formals in the unit above. When instantiating such a generic, clients must specify a type for each generic formal type. The types specified by the client are known as "generic actual types," or "generic actuals" for short. You can remember that by the fact that the actuals are the types "actually" given to the generic unit to work with when instantiated. (You may laugh, but that mnemonic works.)

Now we're ready to discuss the language design concept. The idea is that the syntax of a generic formal type indicates what kind of generic actual is required for a legal instantiation. This is known as the "Contract Model" because we can think of the formal parameters as expressing a contract between the generic unit's implementation and the client code that instantiates the generic. The contract is enforced by the compiler, in that it will reject any instantiation that attempts to specify some actual type that does not match the formal's requirements.

For example, if the generic computes some value for any floating point type, that floating-point type would be declared as a generic formal type, and would be defined so that only

some floating-point type could be used for the corresponding actual type:

```
generic
  type Real is digits <>;
```

The formal parameter syntax reflects the syntax of a floating-point type declaration, except that the <> (the "box") indicates that the generic does not care how many digits are available. The generic actual will be some floating point type and it will specify the number of decimal digits.

If instead we try to match that formal with some actual that is anything other than a floating-point type the compiler will reject the instantiation. Therefore, within the generic body, the implementation code can be written with the assurance that the characteristics and capabilities required of a floating point type will be available. That's the Contract Model in full: the requirements are a matter of the generic unit's purpose and implementation, so the formal parameters reflect those requirements and the compiler ensures they will be met.

Some generic units, though, do not require specifically numeric actual types. These generics can use less specific syntax for their formal types, and as a result, more kinds of actual types are permitted in the instantiations. Remember the Contract Model and this will make sense. The contract between the generic and the clients is, in this case, more permissive: it does not require a numeric type in order to implement whatever it does.

For illustration, suppose we want a generic procedure that will exchange two values of some type. What operations does the generic unit require in the implementation in order to swap two values? There are two: assignment, as you might expect, but also the ability to declare objects of the type (the "temporary" used to hold one of the values during the swap steps). As long as the body can do that, any type will suffice, so the generic formals are written to be that permissive. What is the syntax that expresses that permissiveness, you ask? To answer that, first consider simple, non-generic private types from the user's point of view. For example:

```
package P is
  type Foo is private;
  procedure Do_Something (This : Foo);
private
  type Foo is ... -- whatever
end P;
```

There are two "views" associated with the package: one for the "visible" part of the package spec (declaration), known as the "partial" view, and one for the "private" part of the package spec and the package body, known as the "full" view. The differences between the two views are a function of compile-time visibility.

The partial view is what clients (i.e., users) of the package have: the ability to do things that a type name provides, such as declarations of objects, as well as some basic operations such as assignment, some functions for equality and inequality, some conversions, and whatever subprograms work on the type (the procedure `Do_Something` above). Practically speaking, that's about all that the partial view provides. That's quite a lot, in fact, and corresponds to the classic definition of an "abstract data type."

The code within the package private part and package body has the full view. This code has compile-time visibility to the full definition for type `Foo`, so there are additional capabilities available to this code. For example, if the full definition for `Foo` is as an array type, indexing will be available with the private part and body. If `Foo` is fully defined as some numeric type, arithmetic operations will be possible within the package, and so on.

Therefore, the full view provides capabilities for type `Foo` that users of the type cannot access via the partial view. Only the implementation for type `Foo` and procedure `Do_Something` have the potential to access them.

Now, back to the generic formal parameter. If the generic unit doesn't care what the actual type is, and just needs to be able to do assignment and object declaration, a "generic formal private type" expresses exactly that:

```
generic
  type Item is private;
procedure Exchange( Left, Right : in out Item );

procedure Exchange( Left, Right : in out Item ) is
  Old_Left : Item;
begin
  Old_Left := Left;
  Left := Right;
  Right := Old_Left;
end Exchange;
```

Inside generic procedure `Exchange`, the view of type `Item` is as if `Item` were some private type declared in a package, with only the partial view available. But the operations provided by a partial view are sufficient to implement the body of `Exchange`: only assignment and object declaration are required. Any additional capabilities that the generic actual type may have — array indexing, arithmetic operators, whatever — are immaterial because they are not required. That's the Contract Model: only the specified view's required capabilities are important. Anything else the type can also do is not relevant.

But consider limited types. Those types don't allow assignment, by definition. Therefore, an instantiation that specified a limited actual type for the generic formal type `Item` above would be rejected by the compiler. The contract specifies the ability to do assignment so a limited type would violate the contract.

Finally, as mentioned, our `Exchange` generic needs to declare the "temporary" object `Old_Left`. A partial view of a private type allows that. But not all types are sufficient, by their name alone, to declare objects. Unconstrained array types, such as type **String**, are a familiar example: they require the bounds to be specified when declaring objects; the name **String** alone is insufficient. Therefore, such types would also violate the contract and, therefore, would be rejected by the compiler when attempting to instantiate generic procedure `Exchange`.

Suppose, however, that we have some other generic unit whose implementation does not need to declare objects of the formal type. In that case, a generic actual type that did not support object declaration (by the name alone) would be acceptable for an instantiation. The generic formal syntax for expressing that contract uses these tokens: (`<>`) in addition to the other syntax mentioned earlier:

```
generic
  type Foo(<>) is private;
```

In the above, the generic formal type `Foo` expresses the fact that it can allow unconstrained types — known as "indefinite types" — when instantiated because it will not attempt to use that type name to declare objects. Of course, the compiler will also allow constrained types (e.g., **Integer**, **Boolean**, etc.) in instantiations because it doesn't matter one way or the other inside the generic implementation. The Contract Model says that additional capabilities, declaring objects in this case, are allowed but not required. (There is a way to declare objects of indefinite types, but not using the type name alone. The unchecked facilities don't need to declare objects so we will not show how to do it.)

Now that you understand the Contract Model (perhaps more than you cared), we are ready to examine the generic formal type parameters for `Ada.Unchecked_Conversion`. Here's the declaration again:

```
generic
  type Source(<>) is limited private;
```

(continues on next page)

(continued from previous page)

```

type Target(<>) is limited private;
function Ada.Unchecked_Conversion (S : Source) return Target
with Pure, Nonblocking, Convention => Intrinsic;

```

The two generic formal types, `Source`, and `Target`, are the types used for the incoming value and the returned value, respectively. Both formals are "indefinite, limited private types" in the jargon, but now you know what that means. Inside the implementation of the generic function, neither `Source` nor `Target` will be used to declare objects (the `(<>)` syntax). Likewise, neither type will be used in an assignment statement (the "limited" reserved word). And finally, no particular kind of type is required for `Source` or `Target` (the `private` reserved word). That's a fairly restricted usage within the generic implementation, but as a result the contract can be very permissive: the generic can be instantiated with almost any type. It doesn't matter if the actual is limited or not, private or not, and indefinite or not. The generic implementation doesn't need those capabilities to implement a conversion so they are not part of the contract expressed by the generic formal types.

What sort of type would be disallowed? Abstract types, and incomplete types. However, it is impossible to declare objects of those types, for good reasons, so unchecked conversion is never needed for them.

Note that the result value is returned by-reference whenever possible, in which case it is just a view of the `Source` bits in the formal parameter `S` and not a copy. For a `Source` type that is not a by-copy type, the result of an unchecked conversion will typically be returned by-reference (so that the result and the parameter `S` share the same storage); for a by-copy `Source` type, a copy is made.

The compiler can restrict instantiations but implementers are advised by the language standard to avoid them unless they are required by the target environment. For example, an instantiation for types for which unchecked conversion can't possibly make sense might be disallowed.

Clients can apply language- and vendor-defined restrictions as well, via pragma `Restrictions`. In particular, the language defines the `No_Dependence` restriction, meaning that no client's context clause can specify the unit specified. As a result no client can instantiate the generic for unchecked conversion:

```
pragma Restrictions (No_Dependence => Ada.Unchecked_Conversion);
```

hence there would be no use of unchecked conversion.

From the Contract Model's point of view most any type can be converted to some other type via this generic function. But practically speaking, some limitations are necessary. The following must all be true for the conversion effect to be defined by the language:

- `S'Size = Target'Size`
- `S'Alignment` is a multiple of `Target'Alignment`, or `Target'Alignment` is 0 (meaning no alignment required whatsoever)
- `Target` is not an unconstrained composite type
- `S` and `Target` both have a contiguous representation
- The representation of `S` is a representation of an object of the target subtype

We will examine these requirements in turn, but realize that they are not a matter of legality. Compilers can allow instantiations that violate these requirements. Rather, they are requirements for conversions to have the defined effect.

The first requirement is that the size (in bits) for the parameter `S`, of type `Source`, is the same as the size of the `Target` type. That's reasonable if you consider it. What would it mean to convert, for example, a 32-bit value to an 8-bit value? Which 8 bits should be used?

As a result, one of the few reasons for setting the size of a type (as opposed to the size of an object) is for the sake of well-defined unchecked conversions. We might make the size larger than it would need to be because we want to convert a value of that type to what would otherwise be a larger Target type.

Because converting between types that are not the same size is so open to interpretation, most compilers will issue a warning when the sizes are not the same. Some will even reject the instantiation. GNAT will issue a warning for these cases when the warnings are enabled, but will allow the instantiation. We're supposed to know what we are doing, after all. The warning is enabled via the specific `-gnatwz` switch or the more general `-gnatwa` switch. GNAT tries to be permissive. For example, in the case of discrete types, a shorter source is first zero or sign extended as necessary, and a shorter target is simply truncated on the left. See the GNAT RM for the other details.

The next requirement concerns alignment. As we mentioned earlier, modern architectures tend to have strict alignment requirements. We can meaningfully convert to a type with a stricter alignment, or to a type with no alignment requirement, but converting in the other direction would require a copy.

Next, recall that objects of unconstrained types, such as unconstrained array types or discriminated record types, must have their constraints specified when the objects are declared. We cannot just declare a **String** object, for example, we must also specify the lower and upper bounds. Those bounds are stored in memory, logically as part of the **String** object, since each object could have different bounds (that's the point, after all). What, then, would it mean to convert some value of a type that has no bounds to a type that requires bounds? The third requirement says that it is not meaningful to do so.

The next requirement is that the argument for *S*, and the conversion target type *Target*, have a contiguous representation in memory. In other words, each storage unit must be immediately adjacent, physically, to the next logical storage unit in the value. Such a representation for any given type is not required by the language, although on typical modern architectures it is common. (The type `System.Storage_Elements.Storage_Array` is an exception, in that a contiguous representation is guaranteed.) An instance of `Ada.Unchecked_Conversion` just takes the bits of *S* and treats them as if they are bits for a value of type *Target* (more or less), and does not handle issues of segmentation.

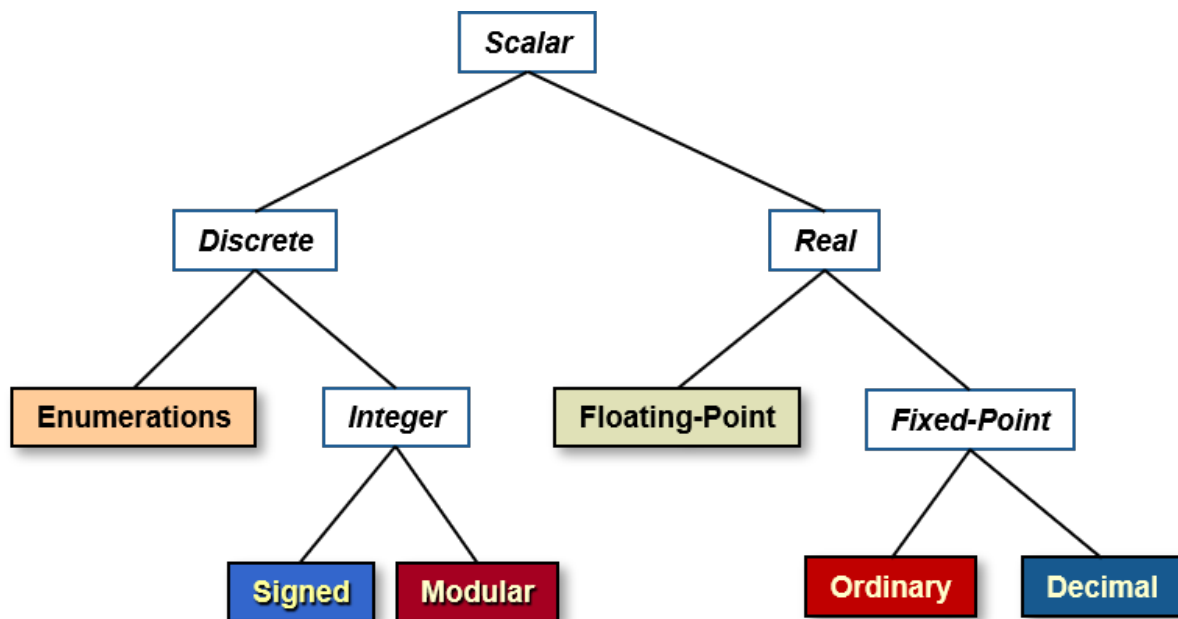
The last requirement merely states that the bits of the argument *S*, when treated as a value of type *Target*, must actually be a bit-pattern representing a value of type *Target* (strictly, the subtype). For example, with signed integers, any bit pattern (of the right size) represents a valid value for those types. In contrast, consider an enumeration type. By default, the underlying representational values are the same as the position values, i.e., starting at zero and increasing by one. But users can override that representation: they can start with any value and, although the values must increase, they need not increase by one:

```
type Toggle_Switch is (Off, On);  
for Toggle_Switch use (Off => 0, On => 4);
```

If we covert an unsigned integer (of the right size) to a `Toggle_Switch` value, what would it mean if the Source value was neither 0 nor 4?

We've said that the instantiations are likely allowed, hence callable functions are created. If the above requirements are not met, what happens?

What happens depends on the Target type, that is, the result type for the conversion. Specifically, it depends on whether the target type is a "scalar" type. As we mentioned earlier, a scalar type is either a "discrete" type or a "real" type, which are themselves further defined, as the figure below indicates. Any other type is a non-scalar type, e.g., record types, access types, task types, and so on.



When the requirements for meaningful instantiations are not respected and the Target type is a scalar type, the result returned from the call is implementation defined and is potentially an invalid representation. For example, type `Toggle_Switch` is an enumeration type, hence it is a scalar type. Therefore, if we covert an unsigned integer (of the right size) to a `Toggle_Switch` value, and the Source value is neither 0 nor 4, the resulting value is an invalid representation. That's the same as an object of type `Toggle_Switch` that is never assigned a value. The random junk in the bits may or may not be a valid `Toggle_Switch` value. That's not a good situation, clearly, but it is well-defined: if it is detected, either `Constraint_Error` or `Program_Error` is raised. If the situation is not detected, execution continues using the invalid representation. In that case it may or may not be detected, near the call or later. For example:

```

with Ada.Unchecked_Conversion;
with Ada.Text_IO; use Ada.Text_IO;
with Interfaces; use Interfaces;

procedure Demo is

  type Toggle_Switch is (Off, On) with Size => 8;
  for Toggle_Switch use (Off => 1, On => 4);

  function As_Toggle_Switch is new Ada.Unchecked_Conversion
    (Source => Unsigned_8, Target => Toggle_Switch);

  T1 : Toggle_Switch;
  T2 : Toggle_Switch;
begin
  T1 := As_Toggle_Switch (12); -- neither 1 nor 4
  if T1 = Off then
    Put_Line ("T1's off");
  else
    Put_Line ("T1's on");
  end if;
  T2 := T1;
  if T2 = Off then
    Put_Line ("T2's off");
  else
    Put_Line ("T2's on");
  end if;
end if;

```

(continues on next page)

(continued from previous page)

```

    Put_Line (T2'Image);
end Demo;

```

In the execution of the code above, the invalid representation value in T1 is not detected, except that it is copied into T2, where it is eventually detected when 'Image is applied to T2. The invalid representation is not detected in the assignment statement or the comparison because we want the optimizer to be able to avoid emitting a check prior to every use of the value. Otherwise the generated code would be too slow. (The language explicitly allows this optimization.)

The evaluation of an object having an invalid representation value due to unchecked conversion is a so-called "bounded error" because the results at run-time are predictable and limited to one of those three possibilities: the two possible exceptions, or continued execution.

Continued execution might even work as hoped, but such code is not portable and should be avoided. A new vendor's compiler, or even a new version of a given vendor's compiler, might detect the situation and raise an exception. That happens, and it ends up costing developer time to make the required application code changes.

The possibilities get much worse when the result type is not a scalar type. In this case, the *effect* of the call — not the value returned by the call — is implementation defined. As a result, the possible run-time behavior is unpredictable and, consequently, from the language rules point of view anything is possible. Such execution is said to be "erroneous."

Why the difference based on scalar versus non-scalar types? Scalar types have a simple representation: their bits directly represent their values. Non-scalar types don't always have a simple representation that can be verified by examining their bits.

For example, we can have record types with discriminants that control the size of the corresponding objects because the record type contains an array component that uses the discriminant to set the upper bound. These record types might have multiple discriminants, and multiple dependent components. As a result, an implementation could have hidden, internal record components. These internal components might be used to store the starting address of the dependent components, for example, or might use pointers to provide a level of indirection. If an unchecked conversion did not provide correct values for these internal components, the effect of referencing the record object would be unpredictable.

Even a comparatively simple record type with one such dependent component is sufficient to illustrate the problem. There are no internal, hidden components involved:

```

with Ada.Unchecked_Conversion;
with Ada.Text_IO;               use Ada.Text_IO;
with System;                    use System; -- for Storage_Unit
with System.Storage_Elements; use System.Storage_Elements;

procedure Demo_Erroneous is

    subtype Buffer_Size is Storage_Offset range 1 .. Storage_Offset'Last;

    type Bounded_Buffer (Capacity : Buffer_Size) is record
        Content : Storage_Array (1 .. Capacity);
        Length : Storage_Offset := 0;
    end record;

    procedure Show_Capacity (This : Bounded_Buffer);

    subtype OneK_Bounded_Buffer is Bounded_Buffer (Capacity => 1 * 1024);

    function As_OneK_Bounded_Buffer is new Ada.Unchecked_Conversion
        (Source => Storage_Array, Target => OneK_Bounded_Buffer);

```

(continues on next page)

(continued from previous page)

```

Buffer    : OneK_Bounded_Buffer;
Sequence  : Storage_Array (1 .. Buffer'Size / Storage_Unit);

procedure Show_Capacity (This : Bounded_Buffer) is
begin
    Put_Line ("This.Capacity is" & This.Capacity'Image);
end Show_Capacity;

begin
    Buffer := As_OneK_Bounded_Buffer (Sequence);
    Put_Line ("Buffer capacity is" & Buffer.Capacity'Image);
    Show_Capacity (Buffer);
    Put_Line ("Done");
end Demo_Erroneous;

```

In the above, the type `Bounded_Buffer` has an array component `Content` that depends on the discriminant `Capacity` for the number of array components. This is an extremely common idiom. However, unchecked conversion is only meaningful, as defined earlier, when converting to constrained target types. `Bounded_Buffer` is not constrained, so we define a constrained subtype (`OneK_Bounded_Buffer`) for the sake of the conversion.

The specific `Buffer` object is 8320 bits ($1024 * 8$, plus $2 * 64$), as is the `Sequence` object, so the sizes are the same.

The alignment of `OneK_Bounded_Buffer` is 8, and `Storage_Array`'s alignment is 1, so the Target type is a multiple of the Source type, as required.

Both types have a contiguous representation, and the sequence of bytes can be a valid representation for the record type, although it certainly might not be valid. For example, if we change the discriminant from what the subtype specifies, we would have an invalid representation for that subtype.

So we can reasonably invoke an unchecked conversion between the array of bytes and the record type. However, as you can see in the code and as the compiler warns, we never assigned a value to the `Sequence` array object. The unchecked conversion from that Sequence of bytes includes the discriminant value, so it is very possible that we will get a discriminant value that is not 1K.

We can test that possibility by running the program. In the first call to `Put_Line`, the program prints the `Capacity` discriminant for the `Buffer` object. The compiler knew it was 1024, so it doesn't get the discriminant component from memory, it just directly prints 1024. However, we can force the compiler to query the discriminant in memory. We can pass `Buffer` to procedure `Show_Capacity`, which takes any `Bounded_Buffer`, and there query (print) the `Capacity` component under that different view. That works because the view inside the procedure `Show_Capacity` is as of `Bounded_Buffer`, in which the discriminant value is unknown at compile-time.

In the above examples, we are responsible for ensuring that the enumeration representation encoding and the record discriminant value are correct when converted from some other type. That's not too hard to recognize because we can literally see in the source code that there is something to be maintained by the conversions. However, there might be hidden implementation artifacts that we cannot see in the source code but that must be maintained nevertheless.

For example, the compiler's implementation for some record type might use dynamic memory allocations instead of directly representing some components. That would not appear in the source code. As a simpler example of invisible implementation issues, consider again our earlier record type:

```
type My_Int is range 1..10;
```

```
subtype S is Integer range 1..10;
```

```
type R is record
```

```
  M : My_Int;
```

```
  X : S;
```

```
  B : Boolean;
```

```
  C : Character;
```

```
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler

3	2	1	0
			M
X	X	X	X
		C	B

R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

As we discussed earlier, between the bytes that are allocated to the record components are some other bytes that are not used at all. As usual, the compiler must implement the language-defined equality operator for the record type. One way to implement that function would be to generate code that checks the equality for each component individually, ignoring any unused bytes. But suppose you have a large record type with many components. The code for checking record level equality will be extensive and inefficient. An alternative implementation for the compiler would be to use a "block compare" machine instruction to check the equality of the entire record at once, rather than component-by-component. That will be considerably more efficient because the block-compare instruction just compares the bits from one starting address to another ending address. But in that case the "unused" bytes are not skipped so the values within those bytes become significant. Comparison of those unused bytes will only work if their values are defined and assigned in each record object. Compilers that may use a block-comparison approach will, therefore, always set those unused bytes to a known value (typically zero). That is part of the valid representation for values of the type, and consequently must be maintained by our unchecked conversions. This being a non-scalar target type, failure to do so results in erroneous execution, i.e., undefined behavior. "There be dragons" as ancient maps of the unknown world once said.

As you can see, you should use unchecked conversions with considerable care and thought. Moreover, because unchecked programming is such a low-level activity, and has vendor-defined implementation issues, it is not only less portable than high-level coding, it is also less portable than other low-level programming. You will be well served if you limit the use of unchecked conversions overall. If your application code is performing unchecked conversions all over the code, something is very likely wrong, or at least very questionable. A well-designed Ada program should not need ubiquitous unchecked conversions.

That said, of course sometimes unchecked conversions are reasonable. But even then, it is better to isolate and hide their use via compile-time visibility controls. For example, instead of having clients invoke unchecked conversion instances many times, have a procedure that is invoked many times, and let the procedure body do the conversion. That way, the clients see a high-level specification of functionality, and, if the conversion needs to be changed later, there is only that one conversion usage (the procedure body) to change. This approach is really just another example of isolating and hiding code that might need to change in the future.

75.7 Data Validity

Our earlier demo program assigned an incorrect value via unchecked conversion into an object of an enumeration type that had non-standard representation values. The value assigned was not one of those representation values so the object had an invalid representation. Certain uses of an invalid representation value will be erroneous, and we saw that the effect of erroneous execution was unpredictable and unbounded.

That example was somewhat artificial, for the sake of illustration. But we might get an

invalid value in a real-world application. For example, we could get an invalid value from a sensor. Hardware sensors are frequently unreliable and noisy. We might get an invalid value from a call to an imported function implemented in some other language. Whenever an assignment is aborted, the target of the assignment might not be fully assigned, leading to so-called "abnormal" values. Other causes are also possible. The problem is not unusual in low-level programming.

How do we avoid the resulting bounded errors and erroneous execution?

In addition to assignment statements, we can safely apply the `Valid` attribute to the object. This language-defined attribute returns a Boolean value indicating whether or not the object's value is a valid representation for the object's subtype. (More details in a moment.) There is no portable alternative to check an object's validity. Here's an example:

```
with Ada.Unchecked_Conversion;
with Ada.Text_IO; use Ada.Text_IO;
with Interfaces; use Interfaces;
with System;

procedure Demo_Validity_Check is

  type Toggle_Switch is (Off, On) with Size => 8;
  for Toggle_Switch use (Off => 1, On => 4);

  T1 : Toggle_Switch;

  function Sensor_Reading (Default : Toggle_Switch) return Toggle_Switch is

    function As_Toggle_Switch is new Ada.Unchecked_Conversion
      (Source => Unsigned_8, Target => Toggle_Switch);

    Result : Toggle_Switch;
    Sensor : Unsigned_8;
    -- for Sensor'Address use System'To_Address (...);

  begin
    Result := As_Toggle_Switch (Sensor);
    return (if Result'Valid then Result else Default);
  end Sensor_Reading;

begin
  T1 := Sensor_Reading (Default => Off); -- arbitrary
  Put_Line (T1'Image);
end Demo_Validity_Check;
```

In the above, `Sensor_Reading` is the high-level, functional API provided to clients. The function hides the use of the unchecked conversion, and also hides the memory-mapped hardware interface named `Sensor`. We've commented out the address clause since we don't really have a memory mapped device available. You can experiment with this program by changing the code to assign a value to `Sensor` (e.g., when it is declared). It is an unsigned 8-bit quantity so any value in the corresponding range would be allowed.

In addition to checking for a valid representation, thus preventing the bounded error, `Valid` also checks that the object is not abnormal, so erroneous execution can be prevented too. (It also checks that any subtype predicate defined for the Target type is also satisfied, but that's a lesson for another day.)

However, the `Valid` attribute can be applied only to scalar objects. There is no language-defined attribute for checking objects of composite types. That's because it would be very hard to implement for some types, if not impossible. For example, given a typical run-time model, it is impossible to check the validity of an access value component. Therefore, you must individually check the validity of scalar record or array components.

At least, you would have to check them individually in standard Ada. GNAT defines another Boolean attribute, named `ValidScalars`, to check them all for us. This attribute returns **True** if the evaluation of `Valid` returns **True** for every scalar subcomponent of the enclosing composite type. It also returns **True** when there are no scalar subcomponents. See the GNAT RM for more information.

MULTI-LANGUAGE DEVELOPMENT

Software projects often involve more than one programming language. Typically that's because there is existing code that already does something we need done and, for that specific code, it doesn't make economic sense to redevelop it in some other language. Consider the rotor blade model in a high-fidelity helicopter simulation. Nobody touches the code for that model except for a few specialists, because the code is extraordinarily complex. (This complexity is unavoidable because a rotor blade's dynamic behavior is so complex. You can't even model it as one physical piece because the tip is traveling so much faster than the other end.) Complex and expensive models like that are a simulator company's crown jewels; their cost is meant to be amortized over as many projects as possible. Nobody would imagine redeveloping it simply because a new project is to be written in a different language.

Therefore, Ada includes extensive facilities to "import" foreign entities into Ada code, and to "export" Ada entities to code in foreign languages. The facilities are so useful that Ada has been used purely as "glue code" to allow code written in two other programming languages to be used together.

You've already seen an introduction to Ada and C code working together in the [*"Interfacing" section of the Ada introductory course*](#) (page 177). If you have not seen that material, be sure to see it first. We will cover some further details not already discussed there, and then go into the details of the facilities not covered elsewhere, but we assume you're familiar with it.

The Ada foreign language interfacing facilities include both "general" and "language-specific" capabilities. The "general" facilities are known as such because they are not tied to any specific language. These pragmas and aspects work with any of the supported foreign languages. In contrast, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. For example, as you saw in that "Interfacing" section, there is a package with a number of declarations for C types, such as **int**, **float**, and **double**, as well as C "strings", with subprograms to convert back and forth between them and Ada's string type. Other languages are also supported, both by the Ada Standard and by vendor additions. You will frequently use both the "general" and the "language-specific" facilities together.

All these interfacing capabilities are defined in Annex B of the language standard. Note that Annex B is not a "Specialized Needs" annex, unlike some of the other annexes. The Specialized Needs annexes are wholly optional, whereas all Ada implementations must implement Annex B. However, some parts of Annex B are optional, so more precisely we should say that every implementation must support all the required features of Annex B. That comes down mainly to the package Interfaces (more on that package in a moment). However, if an implementation does implement any optional part of Annex B, it must be implemented as described by the standard, or with less functionality. An implementation cannot use the same name for some facility (aspect, etc.) but with different semantics. That's true of the Specialized Needs annexes too: not every part need be implemented, but any part that is implemented must conform to the standard. In practice, for Annex B, all implementations provide the required parts, but not all provide support for all the "language-specific" foreign

languages' interfaces. The vendors make a business decision for the optional parts, just as they do regarding the Specialized Needs annexes.

76.1 General Interfacing

In the "Interfacing" section of the Ada introductory course you saw that Ada defines aspects and pragmas for working with foreign languages. These aspects and pragmas are functionally interchangeable, and we will use whichever one of the two that is most convenient in our discussion. The pragmas are officially "obsolescent," but that merely means that a newer approach is available, in this case the corresponding aspects. You can use either one without concern for future support because language constructs that are obsolescent are not removed from the language. Any compiler that supports such constructs will almost certainly support them forever, for the sake of not invalidating existing customers' code. The pragmas have been in the language since Ada 95 so there's a lot of existing code using them. Changing the compiler isn't cost-free, after all, so why spend the money to potentially lose a customer? Likewise, a brand new compiler will also probably support them, for the sake of potentially gaining a customer.

The general interfacing facility consists of these aspects and pragmas, specifically `Import`, `Export`, and `Convention`. As you saw in the Ada Introduction course, `Import` brings a foreign entity into Ada code, `Export` does the opposite, and `Convention` supplies additional information and directives to the compiler. We will go into the details of each.

Regardless of whether the Ada code is importing or exporting some entity, there will be an Ada declaration for that entity. That declaration tells the compiler how the entity can be used, as usual. The interfacing aspects and pragmas are then applied to these Ada declarations.

If we are exporting, then the entity is implemented in Ada. For a subprogram that means there will also be a subprogram body matching the declaration, and the compiler will enforce that requirement as usual. In contrast, if we are importing a subprogram, then it is not implemented in Ada, and therefore there will be no corresponding subprogram body for the Ada declaration. The compiler would not allow it if we tried. In that case the `Import` is the subprogram's completion.

Subprograms often have a separate declaration. Sometimes that's required, for example when we want to include a subprogram as part of a package's API, but at other times it is optional. Remember that a subprogram body acts as a corresponding declaration when there is no separate declaration defined. Thus, either way, we have a subprogram declaration available for the interfacing aspects and/or pragmas.

For data that are imported or exported, we'll have the declaration of the object in Ada to which we can apply the necessary interfacing aspects/pragmas. But we will also have the types for these objects, and as you will see, the types can be part of interfacing too.

76.1.1 Aspect/Pragma Convention

As you saw in the *"Interfacing" section of the Ada introductory course* (page 177), when importing and exporting you'll also specify the "convention" for the entity in question. The pragmas for importing and exporting include a parameter for this purpose. When using the aspects, you'll specify the `Convention` aspect too.

For types, though, you will specify the `Convention` aspect/pragma alone, without `Import` or `Export`. In this case the convention specifies the layout for objects of that type, presumably a layout different than the Ada compiler would normally use. You would need to specify this other layout either because you're going to later declare and export an object of the type, or because you are going to declare an object of the type and pass it as a argument to an imported subprogram.

For example, Ada specifies that multi-dimensional arrays are represented in memory in row-major order. In contrast, the Fortran standard specifies column-major order. If we want to define a type in Ada that can be used for passing parameters to Fortran routines, we need to specify that convention for the type. For example:

```
type Matrix is array (Rows, Columns) of Float
  with Convention => Fortran;
```

(Rows and Columns are user-defined discrete subtypes.)

As a result when we declare Matrix objects the Ada compiler will use the column-major layout. That makes it possible to pass objects of the type to imported Fortran subprograms because the formal parameter will also be of type Matrix. The imported Fortran routine will then see the parameter in memory as it expects to see it. So although you wouldn't need to import or export a type itself, you might very well import or export an object of the type, or pass it as a argument.

When Convention is applied to subprograms, a natural mistake is to think that we are specifying the programming language used to implement the subprogram. In reality, the convention indicates the subprogram calling convention, not the implementation language. The calling convention specifies how parameters are passed to and from subprogram calls, how result values for functions are returned, the order that parameters are pushed on the call stack, how dynamically-sized parameters are passed, and so on. Ordinarily these are matters you don't need to consider because you're working within a single convention automatically, in other words the one used by the Ada compiler you're using.

To illustrate that the convention is not the implementation language, consider a subprogram that we intend to import and call from Ada. This imported routine is implemented in assembly language, but, in addition, let's say it is written to use the same calling convention as the Ada compiler we are using for Ada code. Therefore, the calling convention would be Ada even though the implementation is in assembler.

```
procedure P (X : Integer) with
  ...
  Convention => Ada,
  ...
```

In the example above, Ada is known as a convention identifier, as is Fortran in the earlier example. Convention identifiers are defined by the Ada language standard, but also by Ada vendors.

The Ada standard defines two convention identifiers: Ada (the default), and Intrinsic. In addition, Annex B defines convention identifiers C, COBOL, and Fortran. Support for these Annex B conventions is optional.

GNAT supports the standard and Annex B conventions, as well as the following: Assembler, "C_PLUS_PLUS" (or CPP), Stdcall, WIN32, and a few others. C_PLUS_PLUS is the convention identifier required by the standard when C++ is supported. (Convention identifiers are actual identifiers, not strings, so they must obey the syntax rules for identifiers. "C++" would not be a valid identifier.) See the GNAT User Guide for those other GNAT-specific conventions.

Stdcall and WIN32 actually do specify a particular calling convention, but for those convention identifiers that are language names, how do we get from the name to a calling convention?

The ultimate requirement for any calling convention is compatibility with the Ada compiler we are using. Specifically, the Ada compiler must recognize what the calling convention specifies, and support importing and exporting subprograms with that convention applied.

For the Ada convention that's simple. There is no standard calling convention for Ada. Convention Ada simply means the calling convention applied by the Ada compiler we happen to be using. (We'll talk about Intrinsic shortly.)

So far, so good. But how do we get from those other language names to corresponding calling conventions? There is no standard calling convention for, say, C, any more than there is a standard calling convention for Ada.

In fact we don't get to the calling convention, at least not directly. What the language name in the convention identifier actually tells us is that, when that convention is supported, there is a compiler for that foreign language that uses a calling convention known to, and supported by, the Ada compiler we are using. The Ada compiler vendor defines which languages it supports, after all. For example, when supported, convention C means that there is a compatible C compiler known to the Ada compiler vendor. For GNAT you can guess which C compiler that might be.

It's actually pretty straightforward once you have the big picture. If the convention is supported, the Ada compiler in use knows of a compiler for that language with which it can work. Annex B just defines some convention identifiers for the sake of portability.

But suppose a given Ada compiler supports more than one vendor for a given programming language? In that case the Ada compiler would define and support multiple convention identifiers for the same programming language. Presumably these identifiers would be differentiated by the compiler vendors' names. Thus we might have available conventions GNU_Fortran and Intel_Fortran if both were supported. The Fortran convention identifier would then indicate the default vendor's compiler.

The Intrinsic calling convention represents subprograms that are "built in" to the compiler. When such a subprogram is called the compiler doesn't actually generate the code for an out-of-line call. Instead, the compiler emits the assembly code — often just a single instruction — corresponding to the intrinsic subprogram's name. There will be a separate declaration for the subprogram, but no actual subprogram body containing a sequence of statements. The compiler just knows what to emit in place of the call.

For example:

```
function Shift_Left
  (Value : Unsigned_16;
   Amount : Natural)
return Unsigned_16
with ..., Convention => Intrinsic;
```

The effect is much like a subprogram call that is always in-lined, except that there's no body for the subprogram. In this example the compiler simply issues a shift-left instruction in assembly language.

You'll see the Intrinsic convention applied to many language-defined subprograms. For example:

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target
with ..., Convention => Intrinsic;
```

Thus when we call an instantiation of Ada.Unchecked_Conversion there is no actual call made to some subprogram. The compiler just treats the bits of S as a value of type Target.

Intrinsic subprograms are a good way to access interesting capabilities of the target hardware, without having to write the assembly language yourself (although we will show how to do that, later, directly in Ada). For example, some targets provide an instruction that atomically compares and swaps a value in memory. Ada 2022 just added a standard package for this, but before that we could use the following to access a gcc built-in:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
```

(continues on next page)

(continued from previous page)

```
-- Returns an indication of whether the swap took place.

function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean
with Convention => Intrinsic,
  ...
```

We would specify additional aspects beyond that of Convention but these have not yet been discussed. That's what the ellipses indicate in the various examples above.

76.1.2 Aspect/Pragma Import and Export

You've already seen these aspects in the Ada Introduction course, but for completeness: Import brings a foreign entity into Ada code, and Export makes an Ada entity available to foreign code. In practice, these entities consist of objects and subprograms, but the language doesn't impose many restrictions. It is up to the vendor to decide what makes sense for their specific target.

The aspects Import and Export are so-called Boolean aspects because their value is either **True** or **False**. For example:

```
Obj : Matrix with
  Export => True,
  ...
```

For any Boolean-valued aspect the default is **True** so you only need to give the value explicitly if that value is **False**. There would be no point in doing that in these two cases, of course. Hence we just give the aspect name:

```
Obj : Matrix with
  Export,
  ...
```

Recall that objects of some types are initialized automatically during the objects' elaboration, unless they are explicitly initialized as part of their declarations. Access types are like that, for example. Objects of these types are default initialized to **null** as part of ensuring that their values are always meaningful (absent unchecked conversion).

```
type Reference is access Integer;

Obj : Reference;
```

In the above the value of Obj is **null**, just as if we had explicitly set it that way.

But that initialization is a problem if we are importing an object of an access type. Presumably the value is set by the foreign code, so automatic initialization to null would overwrite the incoming value. Therefore, the language guarantees that implicit initialization won't be applied to imported objects.

```
type Reference is access Integer;

Obj : Reference with Import;
```

Now the value of Obj is whatever the foreign code sets it to, and is not, in other words, overwritten during elaboration of the declaration.

76.1.3 Aspect/Pragma External_Name and Link_Name

For an entity with a **True** Import or Export aspect, we can also specify a so-called external name or link name. These names are specified via aspects `External_Name` and `Link_Name` respectively.

An external name is a string value indicating the name for some entity as known by foreign language code. For an entity that Ada code imports, this is the name that the foreign code declares it to be. For an entity that Ada code exports, this is the name that the foreign code is told to use. This string value is exactly the name to be used, so if you misspell the name the link will fail. For example:

```
function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean
with
  Import,
  Convention   => Intrinsic,
  External_Name => "__sync_bool_compare_and_swap_1";
```

The `External_Name` and `Link_Name` values are strings because the foreign unit names don't necessarily follow the Ada rules for identifiers (the leading underscores in this case). Note that the ending digit in the name above is different from the declared Ada name.

Usually, the name of the imported or exported entity is precisely known and hence exactly specified by `External_Name`. Sometimes, however, a compilation system may have a linker "preprocessor" that augments the name actually used by the linkage step. For example, an implementation might always prepend "_" and then pass the result to the system linker. In that case we don't want to specify the exact name. Instead, we want to provide the "starting point" for the name modification. That's the purpose of the aspect `Link_Name`.

If you don't specify either `External_Name` or `Link_Name` the compilation system will choose one in some implementation-defined manner. Typically this would be the entity's defining name in the Ada declaration, or some simple transformation thereof. But usually we know the name exactly and so we use `External_Name` to give it.

As you can see, it really wouldn't make sense to specify both `External_Name` and `Link_Name` since the semantics of the two conflict. But if both are specified for some reason, the `External_Name` value is ignored.

Note that `Link_Name` cannot be specified for `Intrinsic` subprograms because there is no actual unit being linked into the executable, because intrinsics are built-in. In this case you must specify the `External_Name`.

Finally, because you will see a lot the pragma usage we should go into enough detail so that you know what you're looking at when you see them.

Pragma Import and pragma Export work almost like a subprogram call. Parameters cannot be omitted unless named notation is used. Reordering the parameters is not permitted, however, unlike subprogram calls.

The BNF syntax is as follows. We show Import, but Export has identical parameters:

```
pragma Import(
  [Convention =>] convention_identifier,
  [Entity =>] local_name
  [, [External_Name =>] external_name_string_expression]
  [, [Link_Name =>] link_name_string_expression]);
```

As you can see, the parameters correspond to the individual aspects `Convention`, `External_Name`, and `Link_Name`. When using aspects you don't need to say which Ada entity

you're applying the aspects to, because the aspects are part of the entity declaration syntax. In contrast, the pragma is distinct from the declaration so we must specify what's being imported or exported via the Entity parameter. That's the declared Ada name, in other words. Note that both the External_Name and Link_Name parameters are optional.

Here's that same built-in function, using the pragma to import it:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
-- Returns an indication of whether the swap took place.

function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean;

pragma Import (Intrinsic,
               Sync_Val_Compare_And_Swap_Bool_8,
               "__sync_bool_compare_and_swap_1");
```

The first pragma parameter is for the convention. The next parameter, the Entity, is the Ada unit's declared name. The last parameter is the external name. The compiler either knows what we are referencing by that external name or it will reject the pragma. As we mentioned before, the string value for the name is not required to match the Ada unit name.

You will see later that there are other convention identifiers as well, but we will wait for the [Specific Interfacing section](#) (page 1675) to introduce those.

76.1.4 Package Interfaces

Package Interfaces must be provided by all Ada implementations. The package is intended to provide types that reflect the actual numeric types provided by the target hardware. Of course, the standard has no way to know what hardware is involved, therefore the actual content is implementation-defined. But even so, it is possible to standardize the names for these types, and that is what the language standard does.

Specifically, the standard defines the format for the names for the hardware's signed and modular (unsigned) integer types, and for the floating-point types.

The signed integers have names of the form Integer_n, where *n* is the number of bits used by the machine-supported type. The type for an eight-bit signed integer would be named Integer_8, for example, and then Integer_16 and so on for the larger types, for as many as the target machine supports.

Likewise, for the unsigned integers, the names are of the form Unsigned_n, with the same meaning for *n*. The colloquial eight-bit "byte" would be named Unsigned_8, with Unsigned_16 for the 16-bit version, and so on, again for as many as the machine supports.

For floating-point types it is harder to talk about a format that is sufficiently common to standardize. The IEEE floating-point standard is well known and widely used, however, so if the machine does support the IEEE format that name can be used. Such types would be named IEEE_Float_n, again with the same meaning for *n*. Thus we might see declarations for types IEEE_Float_32 and IEEE_Float_64 and so on, for all the machine supported floating-point types.

In addition to these type declarations, for the unsigned integers only, there will be declarations for shift and rotate operations provided as intrinsic functions.

The resulting package declaration might look something like this:

```
package Interfaces is
```

(continues on next page)

(continued from previous page)

```

type Integer_8 is range -2 ** 7 .. 2 ** 7 - 1;

type Integer_16 is range -2 ** 15 .. 2 ** 15 - 1;

type Integer_32 is range -2 ** 31 .. 2 ** 31 - 1;

...

type Unsigned_8 is mod 2 ** 8;

function Shift_Left (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
function Shift_Right (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
function Rotate_Left (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
function Rotate_Right (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
function Shift_Right_Arithmetic (Value : Unsigned_8; Amount : Natural)
    return Unsigned_8;

type Unsigned_16 is mod 2 ** 16;

function Shift_Left (Value : Unsigned_16; Amount : Natural)
    return Unsigned_16;
function Shift_Right (Value : Unsigned_16; Amount : Natural)
    return Unsigned_16;
...

type Unsigned_32 is mod 2 ** 32;

function Shift_Left (Value : Unsigned_32; Amount : Natural)
    return Unsigned_32;
function Shift_Right (Value : Unsigned_32; Amount : Natural)
    return Unsigned_32;
...

type IEEE_Float_32 is digits 6;
type IEEE_Float_64 is digits 15;
...

end Interfaces;

```

As you can see, when you need to write code in terms of the hardware's numeric types, this package is a great resource. There's no need to declare your own UInt32 type, for example, although of course you could, trivially:

```
type UInt32 is mod 2 ** 32;
```

But if you do, realize that you won't get the shift and rotate operations for your type. Those are only defined for the types in package Interfaces. If you do need to declare such a type, and you do want the additional shift/rotate operations, use inheritance:

```
type UInt32 is new Interfaces.Unsigned_32;
```

GNAT also defines a pragma, as an alternative to inheritance:

```
type UInt32 is mod 2 ** 32;
pragma Provide_Shift_Operators (UInt32);
```

The approach using inheritance is preferable because it is portable, all other things being equal.

One reason to make up your own unsigned type is that you need one that does not in fact reflect the target hardware's numeric types. For example, a hardware device register might have gaps of bits that are currently not used by the device. Those gaps are frequently not

the size of a type declared in package `Interfaces`. We might need an `Unsigned_3` type, for example. That's a reasonable thing to do.

76.2 Language-Specific Interfacing

In addition to the aspects and pragmas for importing and exporting entities that work with any language, Ada also defines standard language-specific facilities for interfacing with a set of foreign languages. The standard defines which languages, but vendors can (and do) expand the set.

Specifically, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. Package `Interfaces` is the root package for a hierarchy of packages that organize these declarations by language, with one or more child packages per language.

Note that the declarations within package `Interfaces` are, by definition, compile-time visible to any child package in the subsystem. Thus whenever one of the language-specific packages needs to mention the machine types they are automatically available.

The standard defines specific support for foreign languages C, COBOL, and Fortran. Thus there are one or more child packages rooted at `Interfaces` that have those language names as their child package names: `Interfaces.C`, `Interfaces.COBOL`, and `Interfaces.Fortran`.

The material below will focus on C and, to a lesser extent, Fortran, ignoring altogether the support for COBOL. That's not because COBOL is unimportant. There is a lot of COBOL business software out there in use. Rather, we skip COBOL because it is not relevant to embedded systems. Similarly, although Fortran is extensively used, especially in high-performance computing, it is not used extensively in embedded systems. We will provide some information about the Fortran support but will not dwell on it.

Even though we do not consider C to be appropriate for large development projects, neither technically not economically, it has its place in small, low-criticality embedded systems. Ada developers can profit from existing device drivers and mature libraries coded in C, for example. Hence interfacing to it is important.

What about C++? Interfacing to C++ is tricky compared to C, because of the vendor-defined name-mangling, automatic invocations of constructors and destructors, exceptions, and so on. Generally, interfacing with C++ code can be facilitated by preventing much of those difficulties using the `extern "C" { ... }` linkage-specification. Doing so then makes the bracketed C++ code look like C, so the C interfacing facilities then can be used.

76.2.1 Package `Interfaces.C`

The child package `Interfaces.C` supports interfacing with units written in the C programming language. Support is in the form of Ada constants and types, and some subprograms. The constants correspond to C's `limits.h` header file, and the Ada types correspond to types for C's `int`, `short`, `unsigned_short`, `unsigned_long`, `unsigned_char`, `size_t`, and so on. There is also support for converting Ada's type `String` to/from `char_array`, and similarly for type `Wide_String`, etc.

It's a large package so we will elide parts. The idea is to give you a feel for what's there. If you want the details, see either the Ada reference manual or bring up the source code in GNAT Studio.

```
package Interfaces.C is
  -- Declaration's based on C's <limits.h>
  CHAR_BIT : constant := 8;
```

(continues on next page)

(continued from previous page)

```

SCHAR_MIN : constant := -128;
SCHAR_MAX : constant := 127;
UCHAR_MAX : constant := 255;

-- Signed and Unsigned Integers. Note that in GNAT, we have ensured that
-- the standard predefined Ada types correspond to the standard C types

type int      is new Integer;
type short   is new Short_Integer;
type long    is range -(2 ** (System.Parameters.long_bits - Integer'(1)))
.. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;
type long_long is new Long_Long_Integer;

type signed_char is range SCHAR_MIN .. SCHAR_MAX;
for signed_char'Size use CHAR_BIT;

type unsigned      is mod 2 ** int'Size;
type unsigned_short is mod 2 ** short'Size;
type unsigned_long  is mod 2 ** long'Size;
type unsigned_long_long is mod 2 ** long_long'Size;

...

-- Floating-Point

type C_float      is new Float;
type double       is new Standard.Long_Float;
type long_double  is new Standard.Long_Long_Float;

-----
-- Characters and Strings --
-----

type char is new Character;

nul : constant char := char'First;

function To_C  (Item : Character) return char;
function To_Ada (Item : char)      return Character;

type char_array is array (size_t range <>) of aliased char;
for char_array'Component_Size use CHAR_BIT;

...

end Interfaces.C;

```

The primary purpose of these types is for use in the formal parameters of Ada subprograms imported from C or exported to C. The various conversion functions can be called from within Ada to manipulate the actual parameters.

When writing the Ada subprogram declaration corresponding to a C function, an Ada procedure directly corresponds to a void function. An Ada procedure also corresponds to a C function if the return value is always to be ignored. Otherwise, the Ada declaration should be a function.

As we said, the types declared in this package can be used as the formal parameter types. That is the intended and recommended approach. However, some Ada types naturally correspond to C types, and you might see them used instead of those from `Interfaces.C`. Type `int` is the C native integer type for the target, for example, as is type `Integer` in Ada. Likewise, C's type `float` and type Ada's `Float` are likely compatible. GNAT goes to some lengths to maintain compatibility with C, since the two gcc compilers share so much internal

technology. Other vendors might not do so. Best practice is use the types in `Interfaces.C` for your parameters.

Of course, the types in `Interfaces.C` are not sufficient for all uses. You will often need to use user-defined types for the formal parameters, such as enumeration types and record types.

Ada enumeration types are compatible with C's enums but note that C requires enum values to be the size of an `int`, whereas Ada does not. The Ada compiler uses whatever sized machine type will support the specified number of enumerational values. It might therefore be smaller than an `int` but it might also be larger. (Declaring more enumeration values than would fit in an integer is unlikely except in tool-generated code, but it is possible.) For example:

```
type Small_Enum is (A, B, C);
```

If we printed the object size for `Small_Enum` we'd get 8 (on a typical machine with GNAT). Therefore, applying the aspect `Convention` to the Ada enumeration type declaration is a good idea:

```
type Small_Enum is (A, B, C) with Convention => C;
```

Now the object size will be 32, the same as `int`.

Speaking of enumeration types, note that Ada 2022 added a boolean type to `Interfaces.C` named `C_Bool` to match that of C99, so you should use it instead of Ada's **Boolean** type for formal parameters.

A simple Ada record type is compatible with a C struct, but remember that the Ada compiler is allowed to reorder the record components. The compiler would do that if it saw that the layout was inefficient, but the point here is that the compiler could do it silently. As a result, you should specify the record layout explicitly using a record representation clause, matching the layout of the C struct in question. Then there will be no question of the layouts matching. Once your record types get more complicated, for example with discriminants or tagged record extensions, things get tricky. Your best bet is to stick with the simple cases when interfacing to C.

Some types that you might think would correspond do not, at least not necessarily. For example, an Ada access type's value might be represented as a simple address, but it might not. In GNAT, an access value designating a value of some unconstrained array type (e.g., **String**) is comprised of two addresses, by default. One designates the characters and the other designates the bounds. You can override that with a pragma, but you must know to do so. For example, if we run the following program, we will see that the object size for the access type `Name` is twice the object size of `System.Address`:

```
with Ada.Text_IO; use Ada.Text_IO;
with System; use System;

procedure Demo is

  type Name is access String;

begin
  Put_Line (Address'Object_Size'Image);
  Put_Line (Name'Object_Size'Image);
end Demo;
```

Some Ada types simply have no corresponding type in C, such as record extensions, task types, and protected types. You'll have to pass those as an "opaque" type, usually as an address. It isn't clear that a C function would know what to do with values of these types, but the general notion of passing an opaque type as an address is useful and not uncommon. Of course, that approach forgoes all type safety, so avoid it when possible.

In addition to the types for the formal parameters, you'll also need to know how parameters are passed to and from C functions. That affects the parameter profiles on both sides, Ada and C. The text in Annex B for Interfaces.C specifies how parameters are to be passed back and forth between Ada and C so that your subprogram declarations can be portable. That's the approach for each supported programming language, i.e., in the discussion of the corresponding child package under Interfaces.

The rules are expressed in terms of scalar types, "elementary" types, array types, and record types. Remember that scalar types are composed of the discrete types and the real types, so we're talking about the signed and modular integers, enumerations, floating-point, and the two kinds of fixed-point types. The "elementary" types consist of the scalars and access types. The rules are fairly intuitive, but throw in Ada's access parameters and parameter modes and some subtleties arise. We won't cover all the various rules but will explore some of the subtleties.

First, the easy cases: mode **in** scalar parameters, such as `int`, as simply passed by copy. Scalar parameters are passed by copy anyway in Ada so the mechanism aligns with C in a straightforward manner. A record type `T` is passed by reference, so on the C side we'd see `t*` where `t` is a C struct corresponding to `T`. A constrained array type in Ada with a component type `T` would correspond to a C formal parameter `t*` where `t` corresponds to `T`. An Ada access parameter **access** `T` corresponds on the C side to `t*` where `t` corresponds to `T`. And finally, a private type is passed according to the full definition of the type; the fact that it is private is just a matter of controlling the client view, being private doesn't affect how it is passed. There are other simple cases, such as access-to-subprogram types, but we can leave that to the Annex.

Now to the more complicated cases. First, some C ABIs (application binary interfaces) pass small structs by copy instead of by reference. That can make sense, in particular when the struct is small, say the size of an address or smaller. In that case there's no performance benefit to be had by passing a reference. When that situation applies, there is another convention we have not yet mentioned: `C_Pass_By_Copy`. As a result the record parameter will be passed by copy instead of the default, by reference (i.e., `T` rather than `*T`), as long as the mode is **in**. For example:

```
type R2 is record
  V : int;
end record
with Convention => C_Pass_By_Copy;

procedure F2 (P : R2) with
  Import,
  Convention => C,
  External_Name => "f2";
```

```
struct R2 {
  int V;
};

void f2 (R2 p);
```

On the C side we expect that `p` is passed by copy and indeed that is how we find it. That said, passing record values to structs by reference is the more common programmer choice. Like arrays, records are typically larger than an address. The point here is that the Ada code can be configured easily to match the C code.

Next, consider passing array values, both to and from C. When passing an array value to C, remember that Ada array types have bounds. Those bounds are either specified at compile time when they are declared, or, for unconstrained array types, specified elsewhere, at run-time.

Array types are not first-class types in C, and C has no notion of unconstrained array types, or even of upper bounds. Therefore, passing an unconstrained array type value is interest-

ing. One approach is to avoid them. Instead, declare a sufficiently large constrained array as a subtype of the unconstrained array type, and then just pass the actual upper bound you want, along with the array object itself.

```
type List is array (Integer range <>) of Interfaces.C.int;

subtype Constrained_List is List (1 .. 100);

procedure P (V : Constrained_List; Size : Interfaces.C.int);
pragma Import (C, P, "p");

Obj : Constrained_List := (others => 42); -- arbitrary values
```

With that, we can just pass the value by reference as usual on the C side:

```
void p (int* v, int size) {
    // whatever
}
```

But that's assuming we know how many array components are sufficient from the C code's point of view. In the example above we'll pass a value up to 100 to the Size parameter and hope that is sufficient.

Really, it would work to use the unconstrained array type as the formal parameter type instead:

```
type List is array (Integer range <>) of Interfaces.C.int;

procedure P (V : List; Size : Interfaces.C.int);
pragma Import (C, P, "p");
```

The C function parameter profile wouldn't change. But why does this work? With values of unconstrained array types, the bounds are stored with the value. Typically they are stored just ahead of the first component, but it is implementation-defined. So why doesn't the above accidentally pass the bounds instead of the first array component itself? It works because we are guaranteed by the Ada language that passing an array will pass (the address of) the components, not the bounds, even for Ada unconstrained array types.

Now for the other direction: passing an array from C to Ada. Here the lack of bounds information on the C side really makes a difference. We can't just pass the array by itself because that would not include the bounds, unlike an Ada call to an Ada routine. In this case the approach is the similar to the first alternative described above, in which we declare a very large array and then pass the bounds explicitly:

```
type List is array (Natural) of int;
-- DO NOT DECLARE AN OBJECT OF THIS TYPE

procedure P (V : List; Size : Interfaces.C.int);
pragma Export (C, P, "p");

procedure P (V : List; Size : Interfaces.C.int) is
begin
    for J in 0 .. Size - 1 loop
        -- whatever
    end loop;
end P;
```

```
extern void p (int* v, int size);

int x [100];

p (x, 100); // call to Ada routine, passing x
```

The fundamental idea is to declare an Ada type big enough to handle anything conceivably needed on the C side. Subtype **Natural** means `0 .. Integer'Last` so `List` is quite large indeed. Just be sure never to declare an object of that type. You'll probably run out of storage on an embedded target.

Earlier we said that it is the Ada type that determines how parameters are passed, and that scalars and elementary types are always passed by copy. For mode **in** that's simple, the copy to the C formal parameter is done and that's all there is to it. But suppose the mode is instead **out** or **in out**? In that case the presumably updated value must be returned to the caller, but C doesn't do that by copy. Here the compiler will come to the rescue and make it work, transparently. Specifically, we just declare the Ada subprogram's formal parameter type as usual, but on the C formal we use a reference. We're talking about scalar and elementary types so let's use `int` arbitrarily. We make the mode **in out** but **out** would also serve:

```
procedure P (Formal : in out int);
```

```
void function p (int* formal);
```

Now the compiler does its magic: it generates code to make a copy of the actual parameter, but it makes that copy into a hidden temporary object. Then, when calling the C routine, it passes the address of the hidden object, which corresponds to the reference expected on the C side. The C code updates the value of the temporary object via the reference, and then, on return, the compiler copies the value back from the temporary to the actual parameter. Problem solved, if a bit circuitous.

There are other aspects to interfacing with C, such as variadic functions that take a varying number of arguments, but you can find these elsewhere in the learn courses.

Next, we examine the child packages under `Interfaces.C`. These packages are not used as much as the parent `Interfaces.C` package so we will provide an overview. You can look up the contents within GNAT Studio or the Ada language standard.

76.2.2 Package `Interfaces.C.Strings`

Package `Interfaces.C` declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type `chars_ptr` corresponds to a common use of `char *` in C programs, and an object of this type can be passed to imported subprograms for which `char *` is the type of the argument of the C function. A subset of the package content is as follows:

```
package Interfaces.C.Strings is
  type chars_ptr is private;
  ...

  function New_Char_Array (Chars : in char_array) return chars_ptr;

  function New_String (Str : in String) return chars_ptr;

  procedure Free (Item : in out chars_ptr);
  ...

  function Value (Item : in chars_ptr) return char_array;
  function Value (Item : in chars_ptr) return String;
  ...

  function Strlen (Item : in chars_ptr) return size_t;
```

(continues on next page)

(continued from previous page)

```

procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars   : in char_array;
                  Check   : in Boolean := True);

...

end Interfaces.C.Strings;

```

Note that allocation might be via `malloc`, or via Ada's allocator `new`. In either case, the returned value is guaranteed to be compatible with `char*`. Deallocation must be via the supplied procedure `Free`.

An amusing point is that you can overwrite the end of the char array just like you can in C, via procedure `Update`. The `Check` parameter indicates whether overwriting past the end is checked. The default is `True`, unlike in C, but you could pass an explicit `False` if you felt the need to do something questionable.

76.2.3 Package Interfaces.C.Pointers

The generic package `Interfaces.C.Pointers` allows us to perform C-style operations on pointers. It includes an access type named `Pointer`, various `Value` functions that dereference a `Pointer` value and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer.

We won't go into the details further. See the Ada RM for more.

76.2.4 Package Interfaces.Fortran

Like `Interfaces.C`, package `Interfaces.Fortran` defines Ada types to be used when working with subprograms using the Fortran calling convention. These types have representations that are identical to the default representations of the Fortran intrinsic types *Integer*, *Real*, *Double Precision*, *Complex*, *Logical*, and *Character* in some supported Fortran implementation. And like the C package, the ways that parameters of various types are passed are also specified.

We leave the details to you to look up in the language standard, if you find them needed in an embedded application.

76.2.5 Machine Code Insertions (MCI)

When working close to the hardware, especially when interacting with a device, it is not uncommon for the hardware to require a very specific set of assembly language instructions to be generated. There are two ways to achieve this: the right way and the wrong way.

The wrong way is to experiment with the source code and compiler switches until you get the exact assembly code you need generated (assuming it is possible at all). But what happens when the next compiler release arrives with a new optimization? And abandon all hope if you go to a new compiler vendor. This approach is both labor-intensive and very brittle.

The right way is to express the precise assembly code sequence explicitly within the Ada source code. (That's true to any high level language, not just Ada.) Or you can call an intrinsic function, if there is one that does exactly what you need. We will focus on inserting it directly, in what is known as "machine code insertion", or "inline assembler."

As an example of the need for this capability, consider the GPIO (General Purpose I/O) port on an STM32 Arm microcontroller. Each port contains 16 individual I/O pins, each of which can be configured as an independent discrete input or output, or as a control line for a

device, with pull-up or pull-down registers, with different clock speeds, and so on. Different on-chip devices use various collections of pins in ways specific to the devices, and require exclusive assignment of the pins. However, any given pin can be used by several different devices. For example, pin 11 on port A ("PA11") can be used by USART #1 as the clear-to-send ("CTS") line, or the CAN #1 bus Rx line, or Channel 4 of Timer 1, among others. Therefore, one of the responsibilities of the system designer is to allocate pins to devices, ensuring that they are allocated uniquely. It is difficult to debug the case in which a pin is accidentally configured for one device and then reconfigured for use with another device (assuming the first device remains in use). To help ensure exclusive allocations, every GPIO port on this Arm implementation has a way of locking the configuration of each I/O pin. That way, some other part of the software can't successfully change the configuration accidentally, for use with some other device. Even if the same configuration was to be used for another device, the lock prevents the accidental update so we find out about the unintentional sharing.

To lock a pin on a port requires a special sequence of reads and writes to a GPIO register for that port. A specific bit pattern is required during the reads and writes. The sequence and bit pattern is such that accidentally locking the pin is highly unlikely.

Once we see how to express assembly language sequences in general we will see how to get the necessary sequence to lock a port/pin pair. Unfortunately, although you can express exactly the code sequence required, such a sequence of assembly language instructions is clearly target hardware-specific. That means portability is inherently limited. Moreover, the syntax for expressing it varies with the vendor, even for the same target hardware. Being able to insert it at the Ada source level doesn't help with either portability issue. You should understand that the use-case for machine code insertion is for small, short sequences. Otherwise you would write the code in assembly language directly, in a separate file. That might obtain a degree of vendor independence, at least for the given target, but not necessarily. The use of inline assembler is intended for cases in which a separate file containing assembly language is not simpler.

With those caveats in place, let's first examine how to do it in general and then how to express it with GNAT specifically.

The right way to express an arbitrary sequence of one or more assembly language statements is to use so-called "code statements." A code statement is an Ada statement, but it is also a qualified expression of a type defined in package `System.Machine_Code`. The content of that package, and the details of code statements, are implementation-defined. Although that affects portability there really is no alternative because we are talking about machine instruction sets, which vary considerably and cannot be standardized at this level.

Package `System.Machine_Code` contains types whose values provide a way of expressing assembly instructions. For example, let's say that there is a "HLT" instruction that halts the processor for some target. There is no other parameter required, just that op-code. Let's also say that one of the types in `System.Machine_Code` is for these "short" instructions consisting only of an op-code. The syntax for the type declaration would then allow the following code statement:

```
Short_Instruction'(Command => HLT);
```

Each of `Short_Instruction`, `Command`, and `HLT` are defined by the vendor in this hypothetical version of package `System.Machine_Code`. You can see why we say that it is both a statement (note the semicolon) and a qualified expression (note the apostrophe).

Code statements must appear in a subprogram body, after the **begin**. Only code statements are allowed in such a body, only use-clauses can be in the declarative part, and no exception handlers are allowed. The complete example would be as follows:

```
procedure Halt -- stops processor
with Inline;
```

(continues on next page)

(continued from previous page)

```

with System.Machine_Code; use System.Machine_Code;
procedure Halt is
begin
  Short_Instruction'(Command => HLT);
end Halt;

```

With that, to halt the processor the Ada code can simply call procedure `Halt`. When the optimizer is enabled there will be no code emitted to make the call, we'd simply see the halt instruction emitted directly in-line.

Package `System.Machine_Code` provides access to machine instructions but as we mentioned, the content is vendor-defined. In addition, the package itself is optional, but is required if Annex C, the Systems Programming Annex, is implemented by the vendor. In practice most all vendors provide this annex.

In GNAT, the content of `System.Machine_Code` looks something like this:

```

type Asm_Input_Operand is ...
type Asm_Output_Operand is ...
type Asm_Input_Operand_List is array (Integer range <>) of Asm_Input_Operand;
type Asm_Output_Operand_List is array (Integer range <>) of Asm_Output_Operand;

type Asm_Insn is private;

...

function Asm
  (Template : String;
   Outputs : Asm_Output_Operand := No_Output_Operands;
   Inputs : Asm_Input_Operand := No_Input_Operands;
   Clobber : String := "";
   Volatile : Boolean := False) return Asm_Insn;

```

With this package content, the expression in a code statement is of type `Asm_Insn`, short for "assembly instruction." Multiple overloaded functions named `Asm` return values of that type.

The `Template` parameter in a string containing one or more assembly language instructions. These instructions are specific to the target machine. The parameter `Outputs` provides mappings from registers to source-level entities that are updated by the assembly statement(s). `Inputs` provides mappings from source-level entities to registers for inputs. `Volatile`, when `True`, tells the compiler not to optimize the call away, and `Clobber` tells the compiler which registers, or memory, if any, are altered by the instructions in `Template`. ("Clobber" is colloquial English for "destroy.") That last is important because the compiler was likely already using some of those registers so the compiler will need to restore them after the call.

We could say, for example, the following, taking all the defaults except for `Volatile`:

```

Asm ("nop", Volatile => True);

```

As you can imagine the full details are extensive, beyond the scope of this introduction. See the GNAT User Guide ("Inline Assembler") for all the gory details.

Now, back to our GPIO port/bin locking example. The port type is declared as follows:

```

type GPIO_Port is limited record
  ...
  LCKR : Word with Atomic; -- lock register
  ...
end record with ...

```

We've elided all but the LCKR component representing the "lock register" within each port. We'd have a record representation clause to ensure the required layout but that's not important here. Word is an unsigned (modular) 32-bit integer type. One of the hardware requirements for accessing the lock register is that the entire register has to be read or written whenever any bits within it are accessed. The compiler must not, for example, write one of the bytes within the register in order to set or clear a bit within that part of the register. Therefore we mark the register as Atomic. If the compiler cannot honor that aspect the compilation will fail, so we would know there is a problem.

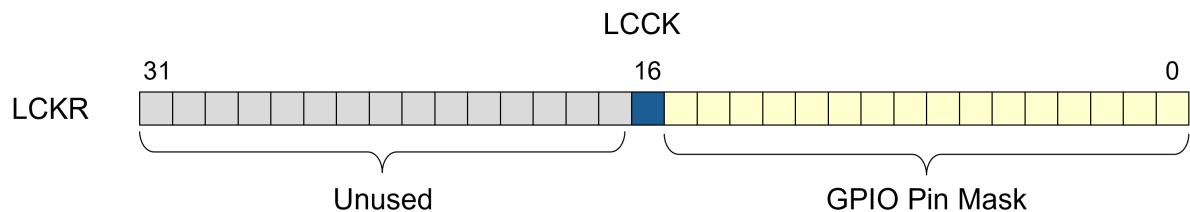
Per the ST Micro Reference Manual, the lock control bit is referred to as LCKK and is bit #16, i.e., the first in the upper half of the LCKR register word.

```
LCKK : constant Word := 16#0001_0000#; -- the "lock control bit"
```

That bit is also known as the "Lock Key" (hence the abbreviation) because it is used to control the locking of port/pin configurations.

There are 16 GPIO pins per port, represented by the lower 16 bits of the register. Each one of these 16 bits corresponds to one of the 16 GPIO pins on a port. If any given bit reads as a 1 then the corresponding pin is locked.

Graphically that looks like this:



Therefore, the Ada types are:

```
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use (Pin_0 => 16#0001#,
                  Pin_1 => 16#0002#,
                  Pin_2 => 16#0004#,
                  ...,
                  Pin_15 => 16#8000#);
```

Note that we had to override the default enumeration representation so that each pin — each enumerational value — would occupy a single dedicated bit in the bit-mask.

With that in place, let's lock a pin. A specific sequence is required to set a pin's lock bit. The sequence writes and reads values from the port's LCKR register. Remember that this 32-bit register has 16 bits for the pin mask (0 .. 15), with bit #16 used as the "lock control bit".

1. write a 1 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
2. write a 0 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
3. do step 1 again
4. read the entire LCKR register
5. read the entire LCKR register again (optional)

Throughout the sequence the same value for the lower 16 bits of the word must be maintained (i.e., the pin mask), including when clearing the LCKK bit in the upper half.

If we wrote this in Ada it would look like this:


```

procedure Lock (Port : in out GPIO_Port; Pin : GPIO_Pin) is
    Temp : Word with Volatile;
begin
    -- set the lock control bit and the pin bit, clear the others
    Temp := LCKR or Pin'Enum_Rep;
    -- write the lock and pin bits
    Port.LCKR := Temp;
    -- clear the lock bit in the upper half
    Port.LCKR := Pin'Enum_Rep;
    -- write the lock bit again
    Port.LCKR := Temp;
    -- read the lock bit
    Temp := Port.LCKR;
    -- read the lock bit again
    Temp := Port.LCKR;
end Lock;

```

Pin'Enum_Rep gives us the underlying value for the enumeration value. We cannot use 'Pos because that attribute provides the logical position number within the enumerated values, and as such always increases consecutively. We need the underlying representation value that we specified explicitly.

The Ada procedure works, but only if the optimizer is enabled (which also precludes debugging). But even so, there is no guarantee that the required assembly language instruction sequence would be generated, especially one that maintains that required bit mask value on each access. A machine-code insertion is appropriate for all the reasons presented earlier:

```

procedure Lock (Port : in out GPIO_Port;
                Pin : GPIO_Pin) is
    use System.Machine_Code, ASCII, System;
begin
    Asm ("orr r3, %1, #65536" & LF & HT & -- 0) Temp := LCKR or Pin'Enum_Rep
        "str r3, [%0, #28]" & LF & HT & -- 1) Port.LCKR := Temp
        "str %1, [%0, #28]" & LF & HT & -- 2) Port.LCKR := Pin'Enum_Rep
        "str r3, [%0, #28]" & LF & HT & -- 3) Port.LCKR := Temp
        "ldr r3, [%0, #28]" & LF & HT & -- 4) Temp := Port.LCKR
        "ldr r3, [%0, #28]" & LF & HT, -- 5) Temp := Port.LCKR
    Inputs => (Address'Asm_Input ("r", This'Address), -- %0
              GPIO_Pin'Asm_Input ("r", Pin)), -- %1
    Volatile => True,
    Clobber => ("r3");
end Lock;

```

We've combined the instructions into one Asm expression. As a result, we can use ASCII line-feed and horizontal tab characters to format the listing produced by the compiler so that each instruction is on a separate line and aligned with the previous instruction, as if we had written the sequence in assembly language directly. That enhances readability later, during examination of the compiler output to verify the required sequence was emitted.

In the above, "%0" is the first input, containing the address of the Port parameter. "%1" is the other input, the value of the Pin parameter. We're using register r3 explicitly, as the "temporary" variable, so we tell the compiler that it has been "clobbered."

If we examine the assembly language output from compiling the file, we find the body of procedure Lock is as hoped:

```

ldr r2, [r0, #4]
ldrh r1, [r0, #8]
.syntax unified
orr r3, r1, #65536
str r3, [r2, #28]

```

(continues on next page)

(continued from previous page)

```
str  r1, [r2, #28]
str  r3, [r2, #28]
ldr  r3, [r2, #28]
ldr  r3, [r2, #28]
```

The first two statements load register 2 (r2) and register 1 (r1) with the subprogram parameters, i.e., the port and pin, respectively. Register 2 gets the starting address of the port record, in particular. (Offset #28 is the location of the LCKR register. The port is passed by reference so that address is actually that of the hardware device.)

We will have separately declared procedure Lock with inlining enabled, so whenever we call the procedure we will get the exact assembly language sequence required to lock the indicated pin on the given port, without any additional code for a procedure call.

Note that we get the calling convention right automatically, because the subprogram is not a foreign entity written in some other language (such as assembly language). It's an Ada subprogram with special content so the Ada convention applies as usual.

76.3 When Ada Is Not the Main Language

When multiple programming languages are involved, the main procedure might not be implemented in Ada. Maybe the bulk of the program is written in C, for example, and this C code calls some Ada routines that have been exported (with the C convention).

That means the Ada builder does not create the executable image's entry point. In fact the Ada main procedure is never the entry point for the final executable image, it's just where the application code begins, like the C main function. There are setup and initialization steps that must happen before any program can execute on a target, and the entry point code is responsible for this functionality. For example, on a bare machine target, the hardware must be initialized, the trap vectors installed, the segments initialized, and so on. On a target running an operating system, the OS is responsible for that initialization but there will be OS-specific initialization steps too. For example, if command-line arguments are supported these may be gathered. All this initialization code is generated by the builder, regardless of the language, followed by a call to the main routine.

Some of the initialization is specific to Ada programming, and must occur before any calls occur to the exported Ada routines. In particular, the entry point code emitted by the Ada builder initializes the Ada run-time system and calls all the elaboration routines for the library units in the application code. Only then does the emitted code invoke the Ada main. If the Ada builder is not going to create the executable it has no chance to emit the code to do that prior initialization. A foreign language builder will not emit such code, so we have a problem.

You could learn enough about how the foreign builder works, and how your Ada builder works, to create a work-around. You could learn what the Ada builder would emit, in other words, and ensure those routines are called manually, either directly or by augmenting the builder scripts (assuming that's possible). But the work-around would be labor-intensive and not robust to changes by the tool vendors. It would be an ugly hack, in other words.

That work-around would not be portable either. The Ada standard can't address hardware- or OS-specific initialization, but it can standardize the name for a routine to do the Ada-specific initialization. Specifically, procedure `adainit` initializes the Ada application code and the Ada run-time library. Similarly, one might need to shut down the Ada code when no further calls will be made to the exported Ada routines. Procedure `adafinal` performs this shut-down functionality. Neither procedure has parameters.

The main function in the other language is intended to import these routines and manually call them each exactly once. `adainit` must be called prior to any calls to the Ada code, and `adafinal` is to be called after all the calls to the Ada code.

For example:

```
#include "stdio.h"

extern int checksum (char *input, int count);

extern void adainit (void);
extern void adafinal (void);

int main (int argc, char *argv[]) {
    char * Str = "Hello World!";
    int sum;
    adainit ();
    sum = checksum (Str, strlen (Str));
    adafinal ();
    printf ("checksum for '%s' is %d", Str, sum);
    return 0;
}
```

In the above, we have an Ada routine to compute a checksum, called by a C main function. Therefore, we use "extern" to tell the C compiler that the "checksum" function is defined elsewhere, i.e., in the Ada routine. Likewise, we tell the compiler that functions `adainit` and `adafinal` are defined elsewhere. The call to `adainit` is made before the call to any Ada code, thus all the elaboration code is guaranteed to happen before checksum needs it. Once the Ada code is not needed, the call to `adafinal` can be made.

Both `adainit` and `adafinal` have no effect after the first invocation. That means you cannot structure your foreign code to iteratively call the two routines whenever you want to invoke some Ada code. In practice you just call them once in the main and be done with it.

INTERACTING WITH DEVICES

Interacting with hardware devices is one of the more frequent activities in embedded systems programming. It is also one of the most enjoyable because you can make something happen in the physical world. There's a reason that making an LED blink is the "hello world" of embedded programming. Not only is it easy to do, it is surprisingly satisfying. I suspect that even the developers of "Full Authority Digital Engine Controllers" (FADEC) — the computers that are in complete, total control of commercial airline engines — have fond memories of making an LED blink early in their careers. And of course a blinking LED is a good way to indicate application status, especially if off-board I/O is limited, which is often the case.

Working at the device register level can be error prone and relatively slow, in terms of source-lines-of-code (SLOC) produced. That's partly because the hardware is in some cases complicated, and partly because of the way the software is written. Using bit masks for setting and clearing bits is not a readable approach, comparatively speaking. There's just not enough information transmitted to the reader. It might be clear enough when written, but will you see it that way months later? Readability is important because programs are read many more times than they are written. Also, an unreadable program is more difficult to maintain, and maintenance is where most money is spent in long-lived applications. Comments can help, until they are out of date. Then they are an active hindrance.

For example, what do you think the following code does? This is real code, where `temp` and `temp2` are unsigned 32-bit integers:

```
temp = ((uint32_t)(GPIO_AF) <<
        ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
        ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
```

That's unfair to ask, absent any context. The code configures a general purpose I/O (GPIO) pin on an Arm microcontroller for one of the "alternate functions". `GPIOx` is a pointer to a GPIO port, `GPIO_PinSource` is a GPIO pin number, and `GPIO_AF` is the alternate function number. But let's say you knew that. Is the code correct? The longer it takes to know, the less productive you are.

The fact that the code above is in C is beside the point. If we wrote it the same way in Ada it would be equally opaque, if not more so. There are simpler approaches. Judicious use of record and array types is one. We'll say more about that later, but the underlying idea is to let the compiler do as much work for us as possible. For example, the data structures used in the code above require explicit shifting whenever they are accessed. If we can avoid that at the source code level — by having the compiler do it for us — we will have simplified the code considerably. Furthermore, letting the compiler do the work for us makes the code more maintainable (which is where the money is). For example, if the code does the shifting explicitly and the data structures are changed, we'll have to change the number of bits to shift left or right. Constants will help there, but we still have to remember to change them;

the compiler won't complain if we forget. In contrast, if we let the compiler do this shifting for us, the amounts to shift will be changed automatically.

Some devices are very simple. In these cases the application may interact directly with the device without unduly affecting productivity. For example, there was a board that had a user-accessible rotary switch with sixteen distinct positions. Users could set the switch to whatever the application code required, e.g., to indicate some configuration information. The entire software interface to this device consisted of a single read-only 8-bit byte in memory. That's all there was to it: you read the memory and thus got the numeric setting of the switch.

More complex devices, however, usually rely on software abstraction to deal with the complexity. Just as abstraction is a fundamental way to combat complexity in software, abstraction also can be used to combat the complexity of driving sophisticated hardware. The abstraction is presented to users by a software "device driver" that exists as a layer between the application code and the hardware device. The layer hides the gory details of the hardware manipulation behind subprograms, types, and parameters.

We say that the device driver layer is an abstraction because, at the least, the names of the procedures and functions indicate what they do, so at the call site you can tell *what* is being done. That's the point of abstraction: it allows us to focus on what, rather than how. Consider that GPIO pin configuration code block again. Instead of writing that block every time we need to configure the alternate function for a pin, suppose we called a function:

```
GPIO_PinAFConfig(USARTx_TX_GPIO_PORT, USARTx_TX_SOURCE, USARTx_TX_AF);
```

The `GPIO_PinAFConfig` function is part of the GPIO device driver provided by the STM32 Standard Peripherals Library (SPL). Even though that's not the best function name conceivable, calls to the function will be far more readable than the code of the body, and we only have to make sure the function implementation is correct once. And assuming the device drivers' subprograms can be inlined, the subprogram call imposes no performance penalty.

Note the first parameter to the call above: `USARTx_TX_GPIO_PORT`. There are multiple GPIO ports on an Arm implementation; the vendor decides how many. In this case one of them has been connected to a USART (Universal Synchronous Asynchronous Receiver Transmitter), an external device for sending and receiving serial data. When there are multiple devices, good software engineering suggests that the device driver present a given device as one of a type. That's what an "abstract data type" (ADT) provides for software and so the device driver applies the same design. An ADT is essentially a class, in class-oriented languages. In Ada, an ADT is represented as a private type declared in a package, along with subprograms that take the type as a parameter.

The Ada Drivers Library (ADL) provided by AdaCore and the Ada community uses this design to supply Ada drivers for the timers, I2C, A/D and D/A converters, and other devices common to microcontrollers. Multiple devices are presented as instances of abstract data types. A variety of development platforms from various vendors are supported, including the STM32 series boards. The library is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library. We are going to use some of these drivers as illustrations in the following sections.

77.1 Non-Memory-Mapped Devices

Some devices are connected to the processor on a dedicated bus that is separate from the memory bus. The Intel processors, for example, used to have (and may still have) instructions for sending and receiving data on this bus. These are the "in" and "out" instructions, and their data-length specific variants.

The original version of Ada defined a package named `Low_Level_IO` for such architectures, but there were very few implementations (maybe just one, known to support the Intel processors). As a result, the package was actually removed from the language standard.

Implementations could still support the package, it just wouldn't be a standard package. That's different from constructs that are marked as "obsolescent" by the standard, e.g., the pragmas replaced by aspects, among other things. Obsolescent constructs are still part of the standard.

If a given target machine has such I/O instructions for the device bus, these can be invoked in Ada via machine-code insertions. For example:

```

procedure Send_Control (Device : Port; Data : Unsigned_16) is
  pragma Suppress (All_Checks);
begin
  asm ("outw %1, (%0)",
      Inputs => (Port'Asm_Input("dx",Device),
                Unsigned_16'Asm_Input("ax",Data)),
      Clobber => "ax, dx");
end Send_Control;

procedure Receive_Control (Device : Port; Data : out Unsigned_16) is
  pragma Suppress (All_Checks);
begin
  asm ("inw (%1), %0",
      Inputs  => (Port'Asm_Input("dx",Device)),
      Outputs => (Unsigned_16'Asm_Output("=ax",Data)),
      Clobber => "ax, dx",
      Volatile => True);
end Receive_Control;

```

Applications could use these subprograms to set the frequency of the Intel PC tone generator, for example, and to turn it on and off. (You can't do that any more in application code because modern operating systems don't give applications direct access to the hardware, at least not by default.)

Although the `Low_Level_IO` package is no longer part of the language, you can write this sort of thing yourself, or vendors can do it. That's possible because the Systems Programming Annex, when implemented, guarantees fully effective use of machine-code inserts. That means you can express anything the compiler could emit. The guarantee is important because otherwise the compiler might "get in the way." For example, absent the guarantee, the compiler would be allowed to insert additional assembly language statements in between yours. That can be a real problem, depending on what your statements do. For instance, if your MCI assembly statements do something and then check a resulting condition code, such as the overflow flag, those interleaved compiler-injected statements might clear that condition code before your code can check it. Fortunately, the annex guarantees that sort of thing cannot happen.

77.2 Memory-Mapped Devices

In *another earlier chapter* (page 1631), we said that we could query the address of some object, and we also showed how to use that result to specify the address of some other object. We used that capability to create an "overlay," in which two objects are used to refer to the same memory locations. As we indicated in that discussion, you would not use the same type for each object — the point, after all, is to provide a view of the shared underlying memory cells that is not already available otherwise. Each distinct type would provide a distinct view of the memory values, that is, a set of operations providing some required functionality.

For example, here's an overlay composed of a 32-bit signed integer object and a 32-bit array object:

```
type Bits32 is array (0 .. 31) of Boolean
  with Component_Size => 1;

X : aliased Integer_32;
Y : Bits32 with Address => X'Address;
```

Because one view is as an integer and the other as an array, we can access that memory using the two different views' operations. Using the view as an array object (Y) we can access individual bits of the memory shared with X. Using the view as an integer (X), we can do arithmetic on the contents of that memory. (We could have used an unsigned integer instead of the signed type, and thereby gained the bit-oriented operations, but that's not the point.)

Very often, though, there is only one Ada object that we place at some specific address. That's because the Ada object is meant to be the software interface to some memory-mapped hardware device. In this scenario we don't have two overlaid Ada objects, we just have one. The other "object" is the hardware device mapped to that starting address. Since they are at the same memory location(s), accessing the Ada object accesses the hardware device.

For a real-world but nonetheless simple example, recall that example of a rotary switch on the front of our embedded computer that we mentioned in the introduction. This switch allows humans to provide some very simple input to the software running on the computer.

```
Rotary_Switch : Unsigned_8 with
  Address => System.Storage_Elements.To_Address (16#FFC0_0801#);
```

We declare the object and also specify the address, but not by querying some entity. We already know the address from the hardware documentation. But we cannot simply use an integer address literal from that documentation because type `System.Address` is almost always a private type. We need a way to compose an `Address` value from an integer value. The package `System.Storage_Elements` defines an integer representation for `Address` values, among other useful things, and a way to convert those integer values to `Address` values. The function `To_Address` does that conversion.

As a result, in the Ada code, reading the value of the variable `Rotary_Switch` reads the number on the actual hardware switch.

Note that if you specify the wrong address, it is hard to say what happens. Likewise, it is an error for an address clause to disobey the object's alignment. The error cannot be detected at compile time, in general, because the address is not necessarily known at compile time. There's no requirement for a run-time check for the sake of efficiency, since efficiency seems paramount here. Consequently, this misuse of address clauses is just like any other misuse of address clauses — execution of the code is erroneous, meaning all bets are off. You need to know what you're doing.

What about writing to the variable? Is that meaningful? In this particular example, no. It is effectively read-only memory. But for some other device it very well could be meaningful, certainly. It depends on the hardware. But in this case, assigning a value to the `Rotary_Switch` variable would have no effect, which could be confusing to programmers. It looks like a variable, after all. We wouldn't declare it as a constant because the human user could rotate the switch, resulting in a different value read. Therefore, we would hide the Ada variable behind a function, precluding the entire issue. Clients of the function can then use it for whatever purpose they require, e.g., as the unique identifier for a computer in a rack.

Let's talk more about the type we use to represent a memory-mapped device. As we said, that type defines the view we have for the object, and hence the operations we have available for accessing the underlying mapped device.

We choose the type for the representative Ada variable based on the interface of the hardware mapped to the memory. If the interface is a single monolithic register, for example,

then an integer (signed or unsigned) of the necessary size will suffice. But suppose the interface is several bytes wide, and some of the bytes have different purposes from the others? In that case, a record type is the obvious solution, with distinct record components dedicated to the different parts of the hardware interface. We could use individual bits too, of course, if that's what the hardware does. Ada is particularly good at this fine-degree of representation because record components of any types can be specified in the layout, down to the bit level, within the record.

In addition, we might want to apply more than one type, at any one time, to a given memory-mapped device. Doing so allows the client code some flexibility, or it might facilitate an internal implementation. For example, the STM32 boards from ST Microelectronics include a 96-bit device unique identifier on each board. The identifier starts at a fixed memory location. In this example we provide two different views — types — for the value. One type provides the identifier as a String containing twelve characters, whereas another type provides the value as an array of three 32-bit unsigned words (i.e., 12 bytes). The two types are applied by two overloaded functions that are distinguished by their return type:

```
package STM32.Device_Id is
    subtype Device_Id_Image is String (1 .. 12);
    function Unique_Id return Device_Id_Image;
    type Device_Id_Tuple is array (1 .. 3) of UInt32
        with Component_Size => 32;
    function Unique_Id return Device_Id_Tuple;
end STM32.Device_Id;
```

The subtype `Device_Id_Image` is the view of the 96-bits as an array of twelve 8-bit characters. (Using type `String` here isn't essential. We could have defined an array of bytes instead of `Character`.) Similarly, subtype `Device_Id_Tuple` is the view of the 96-bits as an array of three 32-bit unsigned integers. Clients can then choose how they want to view the unique id by choosing which function to call.

In the package body we implement the functions as two ways to access the same shared memory:

```
with System;

package body STM32.Device_Id is
    ID_Address : constant System.Address := System.To_Address (16#1FFF_7A10#);

    function Unique_Id return Device_Id_Image is
        Result : Device_Id_Image with Address => ID_Address, Import;
    begin
        return Result;
    end Unique_Id;

    function Unique_Id return Device_Id_Tuple is
        Result : Device_Id_Tuple with Address => ID_Address, Import;
    begin
        return Result;
    end Unique_Id;
end STM32.Device_Id;
```

The GNAT-defined attribute `System.To_Address` in the declaration of `ID_Address` is the same as the function `System.Storage_Elements.To_Address` except that, if the argument is static, the function result is static. This means that such an expression can be used in

contexts (e.g., prelaborable packages) which require a static expression and where the function call could not be used (because the function call is always non-static, even if its argument is static).

The only difference in the bodies is the return type and matching type for the local Result variable. Both functions read from the same location in memory.

Earlier we indicated that the bit-pattern implementation of the GPIO function could be expressed differently, resulting in more readable, therefore maintainable, code. The fact that the code is in C is irrelevant; the same approach in Ada would not be any better. Here's the complete code for the function body:

```
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t       GPIO_AF)
{
    uint32_t temp = 0x00;
    uint32_t temp_2 = 0x00;

    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
    assert_param(IS_GPIO_AF(GPIO_AF));

    temp = ((uint32_t)(GPIO_AF) <<
            ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
    GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
            ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
    temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
    GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}
```

The problem, other than the magic numbers (some named constants would have helped), is that the code is doing nearly all the work instead of off-loading it to the compiler. Partly that's because in C we cannot declare a numeric type representing a 4-bit quantity, so everything is done in terms of machine units, in this case 32-bit unsigned integers.

Why do we need 4-bit values? At the hardware level, each memory-mapped GPIO port has a sequence of 16 4-bit quantities, one for each of the 16 pins on the port. Those 4-bit quantities specify the "alternate functions" that the pin can take on, if needed. The alternate functions allow a given pin to do more than act as a single discrete I/O pin. For example, a pin could be connected to the incoming lines of a USART. We use the configuration routine to apply the specific 4-bit code representing the alternate function required for our application.

These 16 4-bit alternate function fields are contiguous in the register (hence memory) so we can represent them as an array with a total size of 64-bits (i.e., 16 times 4). In the C version this array has two components of type `uint32_t` so it must compute where the corresponding 4-bit value for the pin is located within those two words. In contrast, the Ada version of the array has components of the 4-bit type, rather than two 32-bit components, and simply uses the pin number as the index. The resulting Ada procedure body is extremely simple:

```
procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin
  Port.AFR (Pin) := AF;
end Configure_Alternate_Function;
```

In the Ada version, AFR is a component within the GPIO_Port record type, much like in the C code's struct. However, Ada allows us to declare a much more descriptive set of types,

and it is these types that allows the developer to off-load the work to the compiler.

First, in Ada we can declare a 4-bit numeric type:

```
type Bits_4 is mod 2**4 with Size => 4;
```

The Bits_4 type was already globally defined elsewhere so we just derive our 4-bit "alternate function code" type from it. Doing so allows the compiler to enforce simple strong typing so that the two value spaces are not accidentally mixed. This approach also increases understanding for the reader:

```
type GPIO_Alternate_Function_Code is new Bits_4;
-- We cannot use an enumeration type because there are duplicate binary
-- values
```

Hence type GPIO_Alternate_Function_Code is a copy of Bits_4 in terms of operations and values, but is not the same type as Bits_4 so the compiler will keep them separate for us.

We can then use that type as the array component type for the representation of the AFR:

```
type Alternate_Function_Fields is
  array (GPIO_Pin) of GPIO_Alternate_Function_Code
  with Component_Size => 4, Size => 64; -- both in units of bits
```

Note that we can use the GPIO_Pin parameter directly as the index into the array type, obviating any need to massage the Pin value in the procedure. That's possible because the type GPIO_Pin is an enumeration type:

```
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use
  (Pin_0 => 16#0001#,
   Pin_1 => 16#0002#,
   Pin_2 => 16#0004#,
   Pin_3 => 16#0008#,
   Pin_4 => 16#0010#,
   Pin_5 => 16#0020#,
   Pin_6 => 16#0040#,
   Pin_7 => 16#0080#,
   Pin_8 => 16#0100#,
   Pin_9 => 16#0200#,
   Pin_10 => 16#0400#,
   Pin_11 => 16#0800#,
   Pin_12 => 16#1000#,
   Pin_13 => 16#2000#,
   Pin_14 => 16#4000#,
   Pin_15 => 16#8000#);
```

In the hardware, the GPIO_Pin values don't start at zero and monotonically increase. Instead, the values are bit patterns, where one bit within each value is used. The enumeration representation clause allows us to express that representation.

Type Alternate_Function_Fields is then used to declare the AFR record component in the GPIO_Port record type:

```
type GPIO_Port is limited record
  MODER      : Pin_Modes_Register;
  OTYPER      : Output_Types_Register;
  Reserved_1 : Half_Word;
  OSPEEDR     : Output_Speeds_Register;
```

(continues on next page)

(continued from previous page)

```

PUPDR      : Resistors_Register;
IDR        : Half_Word;          -- input data register
Reserved_2 : Half_Word;
ODR        : Half_Word;          -- output data register
Reserved_3 : Half_Word;
BSRR_Set   : Half_Word;          -- bit set register
BSRR_Reset : Half_Word;          -- bit reset register
LCKR       : Word with Atomic;
AFR        : Alternate_Function_Fields;
Unused     : Unaccessed_Gap;
end record with
  Size => 16#400# * 8;

for GPIO_Port use record
  MODER      at 0  range 0 .. 31;
  OTYPER     at 4  range 0 .. 15;
  Reserved_1 at 6  range 0 .. 15;
  OSPEEDR    at 8  range 0 .. 31;
  PUPDR      at 12 range 0 .. 31;
  IDR        at 16 range 0 .. 15;
  Reserved_2 at 18 range 0 .. 15;
  ODR        at 20 range 0 .. 15;
  Reserved_3 at 22 range 0 .. 15;
  BSRR_Set   at 24 range 0 .. 15;
  BSRR_Reset at 26 range 0 .. 15;
  LCKR       at 28 range 0 .. 31;
  AFR        at 32 range 0 .. 63;
  Unused     at 40 range 0 .. 7871;
end record;

```

These declarations define a record type that matches the content and layout of the STM32 GPIO Port memory-mapped device.

Let's compare the two procedure implementations again. Here they are, for convenience:

```

void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t      GPIO_AF)
{
  uint32_t temp = 0x00;
  uint32_t temp_2 = 0x00;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
  assert_param(IS_GPIO_AF(GPIO_AF));

  temp = ((uint32_t)(GPIO_AF) <<
          ((uint32_t)(GPIO_PinSource & (uint32_t)0x07) * 4));
  GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
          ((uint32_t)(GPIO_PinSource & (uint32_t)0x07) * 4));
  temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
  GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}

```

```

procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin

```

(continues on next page)

(continued from previous page)

```
Port.AFR (Pin) := AF;  
end Configure_Alternate_Function;
```

Which one is correct? Both. But clearly, the Ada version is far simpler, so much so that it is immediately obvious that it is correct. Not so for the coding approach used in the C version, comparatively speaking. It is true that the Ada version required a couple more type declarations, but those make the procedure body far simpler. That resulting simplicity is a reflection of the balance between data structures and executable statements that we should always try to achieve. Ada just makes that easier to achieve than in some other languages.

Of course, the underlying hardware likely has no machine-supported 4-bit unsigned type so larger hardware numeric types are used in the generated code. Hence there are shifts and masking being done in the Ada version as well, but they do not appear in the source code. The developer has let the compiler do that work. An additional benefit of this approach is that the compiler will change the shifting and masking code for us if we change the explicit type declarations.

Why is simplicity so important? Simplicity directly increases understandability, which directly affects correctness and maintainability, which greatly affects the economic cost of the software. In large, long-lived projects, maintenance is by far the largest economic cost driver. In high-integrity applications, correctness is essential. Therefore, doing anything reasonable to keep the code as simple as possible is usually worth the effort. In some projects the non-functional requirements, especially performance, can dictate less simple code, but that won't apply to all of the code. Where possible, simplicity rules.

One more point about the GPIO ports. There are as many of these ports as the Arm microcontroller vendor decides to implement. And as we said, they are memory-mapped, at addresses specified by the vendor. If the memory used by all the ports is contiguous, we can conveniently use an array of the `GPIO_Port` record type to represent all the ports implemented. We would just set the array object's address at the address specified for the first port object in memory. Then, normal array indexing will provide access to any given port in the memory-mapped hardware.

This array approach requires each array component — the `GPIO_Port` record type — to be the right size so that all the array components start on addresses corresponding to the start of the next port in hardware.

That starting address correspondence for the array components is obtained automatically as long as the record type includes all the memory used by any individual device. In that case the next array component will indeed start at an address matching the next device in hardware. Note that this assumes the first array component matches the address of the first hardware device in memory. The first array component is at the same address as the whole array object itself (a fact that is guaranteed by the language), so the array address must be set to whatever the vendor documentation specified for the first port.

However, in some cases the vendor will leave gaps of unused memory for complicated memory-mapped objects like these ports. They do so for the sake of future expansion of the implementation, e.g., to add new features or capacity. The gaps are thus between consecutive hardware devices.

These gaps are presumably (hopefully!) included in the memory layout documented for the device, but it won't be highlighted particularly. You should check, therefore, that the documented starting addresses of the second and subsequent array components are what you will get with a simple array object having components of that record type.

For example, the datasheet for the STM32F407 Arm implementation indicates that the GPIO ports start at address `16#4002_0000#`. That's where `GPIO_A` begins. The next port, `GPIO_B`, starts at address `16#4002_0400#`, or a byte offset of 1024 in decimal. In the STM32F4 Reference Manual, however, the GPIO port register layout indicates a size for any one port that is much less than 1024 bytes. As you saw earlier in the corresponding record

type declaration, on the STM32F4 each port only requires 40 (decimal) bytes. Hence there's a gap of unused memory between the ports, including after the last port, of 984 bytes (7872 bits).

To represent the gap, an "extra", unused record component was added, with the necessary location and size specified within the record type, so that the unused memory is included in the representation. As a result, each array component will start at the right address (again, as long as the first one does). Telling the compiler, and future maintainers, that this extra component is not meant to be referenced by the software would not hurt. You can use the pragma or aspect Unreferenced for that purpose. Here's the code again, for convenience:

```
type GPIO_Port is limited record
  MODER      : Pin_Modes_Register;
  OTYPER      : Output_Types_Register;
  Reserved_1  : Half_Word;
  OSPEEDR     : Output_Speeds_Register;
  PUPDR       : Resistors_Register;
  IDR         : Half_Word;           -- input data register
  Reserved_2  : Half_Word;
  ODR         : Half_Word;           -- output data register
  Reserved_3  : Half_Word;
  BSRR_Set    : Half_Word;           -- bit set register
  BSRR_Reset  : Half_Word;           -- bit reset register
  LCKR        : Word with Atomic;
  AFR         : Alternate_Function_Fields;
  Unused      : Unaccessed_Gap with Unreferenced;
end record with
  Size => 16#400# * 8;

for GPIO_Port use record
  MODER      at 0  range 0 .. 31;
  OTYPER      at 4  range 0 .. 15;
  Reserved_1  at 6  range 0 .. 15;
  OSPEEDR     at 8  range 0 .. 31;
  PUPDR       at 12 range 0 .. 31;
  IDR         at 16 range 0 .. 15;
  Reserved_2  at 18 range 0 .. 15;
  ODR         at 20 range 0 .. 15;
  Reserved_3  at 22 range 0 .. 15;
  BSRR_Set    at 24 range 0 .. 15;
  BSRR_Reset  at 26 range 0 .. 15;
  LCKR        at 28 range 0 .. 31;
  AFR         at 32 range 0 .. 63;
  Unused      at 40 range 0 .. 7871;
end record;
```

The type for the gap, Unaccessed_Gap, must represent 984 bytes so we declared an array like so:

```
Gap_Size : constant := 984; -- bytes
-- There is a gap of unused, reserved memory after the end of the
-- bytes used by any given memory-mapped GPIO port. The size of the
-- gap is indicated in the STM32F405xx etc. Reference Manual, RM 0090.
-- Specifically, Table 1 shows the starting and ending addresses mapped
-- to the GPIO ports, for an allocated size of 16#400#, or 1024 (decimal)
-- bytes per port. However, in the same document, the register map for
-- these ports shows only 40 bytes currently in use. Presumably this gap is
-- for future expansion when additional functionality or capacity is added,
-- such as more pins per port.

type Unaccessed_Gap is array (1 .. Gap_Size) of Unsigned_8 with
  Component_Size => Unsigned_8'Size,
```

(continues on next page)

(continued from previous page)

```

Size          => Gap_Size * Unsigned_8'Size;
-- This type is used to represent the necessary gaps between GPIO
-- ports in memory. We explicitly allocate a record component of
-- this type at the end of the record type for that purpose.

```

We also set the size of the entire record type to `16#400#` bytes since that is the total of the required bytes plus the gap, as per the documentation. As such, this is a "confirming" size clause because the reserved gap component increases the required size to that value (which is the point). We don't really need to do both, i.e., declare the reserved gap component and also set the record type size to the larger value. We could have done either one alone. One could argue that setting the size alone would have been simpler, in that it would obviate the type declaration and corresponding record component declaration. Being doubly explicit seemed a good idea at the time.

77.3 Dynamic Address Conversion

In the overlay example there were two distinct Ada objects, of two different types, sharing one (starting) address. The overlay provides two views of the memory at that address because there are two types involved. In this idiom the address is known when the code is written, either because it is a literal value specified in some hardware spec, or it is simply the address of the other object (in which case the actual address value is neither known nor relevant).

When there are several views required, declaring multiple overlaid variables at the same address absolutely can work, but can be less convenient than an alternative idiom. The alternative is to convert an address value to a value of an access type. Dereferencing the resulting access value provides a view of the memory corresponding to the designated type, starting at the converted address value.

For example, perhaps a networking component is given a buffer — an array of bytes — representing a received message. A subprogram is called with the buffer as a parameter, or the parameter can be the address of the buffer. If the subprogram must interpret this array via different views, this alternative approach works well. We could have an access type designating a message preamble, for example, and convert the first byte's address into such an access value. Dereferencing the conversion gives the preamble value. Likewise, the subprogram might need to compute a checksum over some of the bytes, so a different view, one of an array of a certain set size, could be used. Again, we could do that with overlaid objects but the alternative can be more convenient.

Here's a simple concrete example to illustrate the approach. Suppose we want to have a utility to swap the two bytes at any arbitrary address. Here's the declaration:

```

procedure Swap2 (Location : System.Address);

```

Callers pass the address of an object intended to have its (first) two bytes swapped:

```

Swap2 (Z'Address);

```

In the call, Z is of type `Interfaces.Integer_16`, for example, or `Unsigned_16`, or even something bigger as long as you only care about swapping the first two bytes.

The incomplete implementation using the conversion idiom could be like so:

```

procedure Swap2 (Location : System.Address) is
  X : Word renames To_Pointer (Location).all;
begin
  X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;

```

The declaration of X is the pertinent part.

In the declaration, X is of type Word, a type (not yet shown) derived from Interfaces.Unsigned_16. Hence X can have the inherited shift and logical **or** operations applied.

The To_Pointer (Location) part of the declaration is a function call. The function returns the conversion of the incoming address value in Location into an access value designating Word values. We'll explain how to do that momentarily. The **.all** explicitly dereferences the access value resulting from the function call.

Finally, X renames the Word designated by the converted access value. The benefit of the renaming, in addition to the simpler name, is that the function is only called once, and the access value deference is only evaluated once.

Now for the rest of the implementation not shown earlier.

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;
```

System.Address_To_Access_Conversions is a language-defined generic package that provides just two functions: one to convert an address value to an access type, and one to convert in the opposite direction:

```
generic
  type Object (<=>) is limited private;
package System.Address_To_Access_Conversions is

  type Object_Pointer is access all Object;

  function To_Pointer (Value : Address) return Object_Pointer;
  function To_Address (Value : Object_Pointer) return Address;

  pragma Convention (Intrinsic, To_Pointer);
  pragma Convention (Intrinsic, To_Address);

end System.Address_To_Access_Conversions;
```

Object is the generic formal type parameter, i.e., the type we want our converted addresses to designate via the type Object_Pointer. In the byte-swapping example, the type Word was passed to Object in the instantiation.

The access type used by the functions is Object_Pointer, declared along with the functions. Object_Pointer designates values of the type used for the generic actual parameter, in this case Word.

Note the pragma Convention applied to each function, indicating that there is no actual function call involved; the compiler emits the code directly, if any code is actually required. Otherwise the compiler just treats the incoming **Address** bits as a value of type Object_Pointer.

The instantiation specifies type Word as the generic actual type parameter, so now we have a set of functions for that type, in particular To_Pointer.

Let's look at the code again, this time with the additional declarations:

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;

procedure Swap2 (Location : System.Address) is
  X : Word renames To_Pointer(Location).all;
```

(continues on next page)

(continued from previous page)

```

begin
  X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;

```

Word_Ops is the generic instance, followed immediately by a **use** clause so that we can refer to the visible content of the package instance conveniently.

In the renaming expression, To_Pointer (Location) converts the incoming address in Location to a pointer designating the Word at that address. The **.all** dereferences the resulting access value to get the designated Word value. Hence X refers to that two-byte value in memory.

We could almost certainly achieve the same affect by replacing the call to the function in To_Pointer with a call to an instance of Ada.Unchecked_Conversion. The conversion would still be between an access type and a value of type System.Address, but the access type would require declaration by the user. In both cases there would be an instantiation of a language-defined facility, so there's not much saving in lines of source code, other than the access type declaration. Because System.Address_To_Access_Conversions is explicitly intended for this purpose, good style suggests its use in preference to unchecked conversion, but both approaches are common in production code.

In either case, the conversion is not required to work, although in practice it will, most of the time. Representing an access value as an address value is quite common because it matches the typical underlying hardware's memory model. But even so, a single address is not necessarily sufficient to represent an access value for any given designated type. In that case problems arise, and they are difficult to debug.

For example, in GNAT, access values designating values of unconstrained array types, such as **String**, are represented as two addresses, known as "fat pointers". One address points to the bounds for the specific array object, since they can vary. The other address designates the characters. Therefore, conversions of a single address to an access value requiring fat pointers will not work using unchecked conversions. (There is a way, however, to tell GNAT to use a single address value, but it is an explicit step in the code. Once done, though, unchecked conversions would then work correctly.)

You can alternatively use generic package System.Address_To_Access_Conversions. That generic is defined for the purpose of converting addresses to access values, and vice versa. But note that the implementation of the generic's routines must account for the representation their compiler uses for unbounded types like **String**.

77.4 Address Arithmetic

Part of "letting the compiler do the work for you" is not doing address arithmetic in the source code if you can avoid it. Instead, for instance, use the normal "dot notation" to reference components, and let the compiler compute the offsets to those components. The approach to implementing procedure Configure_Alternate_Function for a GPIO_Port is a good example.

That said, sometimes address arithmetic is the most direct expression of what you're trying to implement. For example, when implementing your own memory allocator, you'll need to do address arithmetic.

Earlier in this section we mentioned the package System.Storage_Elements, for the sake of the function that converts integer values to address values. The package also defines functions that provide address arithmetic. These functions work in terms of type System.Address and the package-defined type Storage_Offset. The type Storage_Offset is an integer type with an implementation-defined range. As a result you can have positive and negative offsets, as needed. Addition and subtraction of offsets to/from addresses is supported, as well as the **mod** operator.

Combined with package `System` (for type `System.Address`), the functions and types in this package provide the kinds of address arithmetic other languages provide. Nevertheless, you should prefer having the compiler do these computations for you, if possible.

Here's an example illustrating the facilities. The procedure defines an array of record values, then traverses the array, printing the array components as it goes. (This is not the way to really implement such code. It's just an illustration for address arithmetic.)

```
with Ada.Text_IO;           use Ada.Text_IO;
with System.Storage_Elements; use System.Storage_Elements;
with System.Address_To_Access_Conversions;

procedure Demo_Address_Arithmetic is

  type R is record
    X : Integer;
    Y : Integer;
  end record;

  R_Size : constant Storage_Offset := R'Object_Size / System.Storage_Unit;

  Objects : aliased array (1 .. 10) of aliased R;    -- arbitrary bounds

  Objects_Base : constant System.Address := Objects'Address;

  Offset : Storage_Offset;

  -- display the object of type R at the address specified by Location
  procedure Display_R (Location : in System.Address) is

    package R_Pointers is new System.Address_To_Access_Conversions (R);
    use R_Pointers;

    Value : R renames To_Pointer (Location).all;
    -- The above converts the address to a pointer designating an R value
    -- and dereferences it, using the name Value to refer to the
    -- dereferenced R value.
  begin
    Put (Integer'Image (Value.X));
    Put (" ", " ");
    Put (Integer'Image (Value.Y));
    New_Line;
  end Display_R;

begin
  Objects := ((0,0), (1,1), (2,2), (3,3), (4,4),
             (5,5), (6,6), (7,7), (8,8), (9,9));

  Offset := 0;

  -- walk the array of R objects, displaying each one individually by
  -- adding the offset to the base address of the array
  for K in Objects'Range loop
    Display_R (Objects_Base + Offset);
    Offset := Offset + R_Size;
  end loop;
end Demo_Address_Arithmetic;
```

Seriously, this is just for the purpose of illustration. It would be much better to just index into the array directly.

GENERAL-PURPOSE CODE GENERATORS

In [another chapter](#) (page 1667), we mentioned that the best way to get a specific set of machine instructions emitted from the compiler is to write them ourselves, in the Ada source code, using machine-code insertions (MCI). The rationale was that the code generator will make reasonable assumptions, including the assumption that performance is of uppermost importance, but that these assumptions can conflict with device requirements.

For example, the code generator might not issue the specific sequence of machine code instructions required by the hardware. The GPIO pin "lock" sequence in that referenced chapter is a good example. Similarly, the optimizer might remove what would otherwise be "redundant" read/writes to a memory-mapped variable.

The code generator might issue instructions to read a small field in a memory-mapped record object using byte-sized accesses, when instead the device requires whole-word or half-word access instructions.

The code generator might decide to load a variable from memory into a register, accessing the register when the value is required. Typically that approach will yield far better performance than going to memory every time the value is read or updated. But suppose the variable is for a memory-mapped device? In that case we really need the generated code to go to memory every time.

As you can see, there are times when we cannot let the code generator make the usual assumptions. Therefore, Ada provides aspects and pragmas that developers can use to inform the compiler of facts that affect code generation in this regard.

These facilities are defined in the Systems Programming Annex, C.6, specifically. The title of that sub-clause is "Shared Variables" because the objects (memory) can be shared between tasks as well as between hardware devices and the host computer. We ignore the context of variables shared between tasks, focusing instead of shared memory-mapped devices, as this course is about embedded systems.

When describing these facilities we will use aspects, but remember that the corresponding pragmas are defined as well, except for one. (We'll mention it later.) For the other aspects, the pragmas existed first and, although obsolescent, remain part of the language and supported. There's no need to change your existing source code using the pragmas to use the aspects instead, unless you need to change it for some other reason.

As this is an introduction, we will not go into absolutely all the details, but will instead give a sense of what the language provides, and why.

78.1 Aspect Independent

To interface with a memory-mapped device, there will be an Ada object of an appropriate type that is mapped to one or more bytes of memory. The software interacts with the device by reading and/or writing to the memory locations mapped to the device, using the operations defined by the type in terms of normal Ada semantics.

Some memory-mapped devices can be directly represented by a single scalar value, usually of some signed or unsigned numeric type. More sophisticated devices almost always involve several distinct input and output fields. Therefore, representation in the software as a record object is very common. Ada record types have such extensive and flexible support for controlling their representation, down to the individual bit level, that using a record type makes sense. (And as mentioned, using normal record component access via the "dot notation" offloads to the compiler the address arithmetic needed to access individual memory locations mapped to the device.) And of course the components of the mapped record type can themselves be of scalar and composite types too, so an extensive descriptive capability exists with Ada.

Let's say that one of these record components is smaller than the size of the smallest addressable memory unit on the machine, which is to say, smaller than the machine instructions can read/write memory individually. A Boolean record component is a good example, and very common. The machine cannot usually read/write single bits in memory, so the generated code will almost certainly read or write a byte to get the enclosed single-bit Boolean component. It might use a larger sized access too, a half-word or word. Then the generated code masks off the bits that are not of interest and does some shifts to get the desired component.

Reading and writing the bytes surrounding the component accessed in the source code can cause a problem. In particular, some devices react to being read or written by doing something physical in the hardware. That's the device designer's intent for the software. But we don't want that to happen accidentally due to surrounding bytes being accessed.

Therefore, to prevent these "extra" bytes from being accessed, we need a way to tell the compiler that we need the read or write accesses for the given object to be independent of the surrounding memory. If the compiler cannot do so, we'll get an error and the compilation will fail. That beats debugging, every time.

Therefore, the aspect `Independent` specifies that the code generated by the compiler must be able to load and store the memory for the specified object without also accessing surrounding memory. More completely, it declares that a type, object, or component must be independently addressable by the hardware. If applied to a type, it applies to all objects of the type.

Likewise, aspect `Independent_Components` declares that the individual components of an array or record type must be independently addressable.

With either aspect the compiler will reject the declaration if independent access is not possible for the type/object in question.

For example, if we try to mark each Boolean component of a record type as `Independent` we can do so, either individually or via `Independent_Components`, but doing so will require that each component is a byte in size (or whatever the smallest addressable unit happens to be on this machine). We cannot make each Boolean component occupy one bit within a given byte if we want them to be independently accessed.

```
package P is
  type R is record
    B0 : Boolean;
    B1 : Boolean;
    B2 : Boolean;
    B3 : Boolean;
    B4 : Boolean;
    B5 : Boolean;
  end record with
    Size => 8,
    Independent_Components;

  for R use record
```

(continues on next page)

(continued from previous page)

```

    B0 at 0 range 0 .. 0;
    B1 at 0 range 1 .. 1;
    B2 at 0 range 2 .. 2;
    B3 at 0 range 3 .. 3;
    B4 at 0 range 4 .. 4;
    B5 at 0 range 5 .. 5;
end record;

end P;
```

For a typical target machine the compiler will reject that code, complaining that the Size for R' must be at least 48 bits, i.e., 8 bits per component. That's because the smallest quantity this machine can independently address is an 8-bit byte.

But if we don't really need the individual bits to be independently accessed — and let's hope no hardware designer would define such a device — then we have more flexibility. We could, for example, require that objects of the entire record type be independently accessible:

```

package Q is

  type R is record
    B0 : Boolean;
    B1 : Boolean;
    B2 : Boolean;
    B3 : Boolean;
    B4 : Boolean;
    B5 : Boolean;
  end record with
    Size => 8,
    Independent;

  for R use record
    B0 at 0 range 0 .. 0;
    B1 at 0 range 1 .. 1;
    B2 at 0 range 2 .. 2;
    B3 at 0 range 3 .. 3;
    B4 at 0 range 4 .. 4;
    B5 at 0 range 5 .. 5;
  end record;

end Q;
```

This the compiler should accept, assuming a machine that can access bytes in memory individually, without having to read some number of other bytes.

But for another twist, suppose we need one of the components to be aliased, so that we can construct access values designating it via the **Access** attribute? For example, given the record type R above, and some object Foo of that type, suppose we want to say Foo.B0'Access? We'd need to mark the component as **aliased**:

```

package QQ is

  type R is record
    B0 : aliased Boolean;
    B1 : Boolean;
    B2 : Boolean;
    B3 : Boolean;
    B4 : Boolean;
    B5 : Boolean;
  end record with
    Size => 8,
```

(continues on next page)

(continued from previous page)

```
Independent;  
  
for R use record  
  B0 at 0 range 0 .. 0;  
  B1 at 0 range 1 .. 1;  
  B2 at 0 range 2 .. 2;  
  B3 at 0 range 3 .. 3;  
  B4 at 0 range 4 .. 4;  
  B5 at 0 range 5 .. 5;  
end record;  
  
end QQ;
```

The compiler will once again reject the code, complaining that the size of B0 must be a multiple of a `Storage_Unit`, in other words, the size of something independently accessible in memory on this machine.

Why? The issue here is that aliased objects, including components of composite types, must be represented in such a way that creating the designating access ("pointer") value is possible. The component B0, if allocated only one bit, would not allow an access value to be created due to the usual machine accessibility limitation we've been discussing.

Similarly, a record component that is of some by-reference type, such as any tagged type, introduces the same issues as an aliased component. That's because the underlying implementation of by-reference parameter passing is much like a '[Access](#)' attribute reference.

As important as the effect of this aspect is, you probably won't see it specified. There are other aspects that are more typically required. However, the semantics of `Independent` are part of the semantics of some of these other aspects. Applying them applies `Independent` too, in effect. So even though you don't typically apply it directly, you need to understand the independent access semantics. We discuss these other, more commonly applied aspects next.

These representation aspects may be specified for an object declaration, a component declaration, a full type declaration, or a generic formal (complete) type declaration. If any of these aspects are specified `True` for a type, then the corresponding aspect is `True` for all objects of the type.

78.2 Aspect Volatile

Earlier we said that the compiler (specifically the optimizer) might decide to load a variable from memory into a register, accessing the register when the value is required or updated. Similarly, the compiler might reorder instructions, and remove instructions corresponding to redundant assignments in the source code. Ordinarily we'd want those optimizations, but in the context of embedded memory-mapped devices they can be problematic.

The hardware might indeed require the source code to read or write to the device in a way that the optimizer would consider redundant, and in order to interact with the device we need every read and write to go to the actual memory for the mapped device, rather than a register. As developers we have knowledge about the context that the compiler lacks.

The compiler is aware of the fact that the Ada object is memory-mapped because of the address clause placing the object at a specific address. But the compiler does not know we are interacting with an external hardware device. Perhaps, instead, the object is mapped to a specific location because some software written in another language expects to access it there. In that case redundant reads or writes of the same object really would be redundant. The fact that we are interacting with a hardware device makes a difference.

In terms of the language rules, we need reading from, and writing to, such devices to be part of what the language refers to as the "external effects" of the software. These effects

are what the code must actually produce. Anything else — the internal effects — could be removed by the optimizer.

For example, suppose you have a program that writes a value to some variable and also writes the string literal "42" to a file. That's is absolutely all that the program contains.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
  Output : File_Type;
  Silly   : Integer;
begin
  Silly := 0;
  Create (Output, Out_File, "output.txt");
  Put (Output, "42");
  Close (Output);
end Demo;
```

The value of the variable `Silly` is not used in any way so there is no point in even declaring the variable, much less generating code to implement the assignment. The update to the variable has only an internal effect. With warnings enabled we'll receive notice from the compiler, but they're just warnings.

However, writing to the file is an external effect because the file persists beyond the end of the program's execution. The optimizer (when enabled) would be free to remove any access to the variable `Silly`, but not the write to the file.

We can make the compiler recognize that a software object is part of an external effect by applying the aspect `Volatile`. (Aspect `Atomic` is pertinent too. More in a moment.) As a result, the compiler will generate memory load or store instructions for every read or update to the object that occurs in the source code. Furthermore, it cannot generate any additional loads or stores to that variable, and it cannot reorder loads or stores from their order in the source code. "What You See Is What You Get" in other words.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Demo is
  Output : File_Type;
  Silly   : Integer with Volatile;
begin
  Silly := 0;
  Create (Output, Out_File, "output.txt");
  Put (Output, "42");
  Close (Output);
end Demo;
```

If we compile the above, we won't get the warning we got earlier because the compiler is now required to generate the assignment for `Silly`.

The variable `Silly` is not even a memory-mapped object, but remember that we said these aspects are important to the tasking context too, for shared variables. We're ignoring that context in this course.

There is another reason to mark a variable as `Volatile`. Sometimes you want to have exactly the load and store instructions generated that match those of the Ada code, even though the volatile object is not a memory-mapped object. For example, [elsewhere](#) (page 1667) we said that the best way to achieve exact assembly instruction sequences is the use of machine-code inserts (MCI). That's true, but for the moment let's say we want to write it in Ada without the MCIs. Our earlier example was the memory-mapped GPIO ports on Arm microcontrollers produced by ST Microelectronics. Specifically, these ports have a "lock" per GPIO pin that allows the developer to configure the pin and then lock it so that no other configuration can accidentally change the configuration of that pin. Doing so requires an exact sequence of loads and stores. If we wrote this in Ada it would look like this:

```
procedure Lock
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin)
is
  Temp : Word with Volatile;
begin
  -- set the lock control bit and the pin
  -- bit, clear the others
  Temp := LCKR or Pin'Enum_Rep;

  -- write the lock and pin bits
  Port.LCKR := Temp;

  -- clear the lock bit in the upper half
  Port.LCKR := Pin'Enum_Rep;

  -- write the lock bit again
  Port.LCKR := Temp;

  -- read the lock bit
  Temp := Port.LCKR;

  -- read the lock bit again
  Temp := Port.LCKR;
end Lock;
```

Temp is marked volatile for the sake of getting exactly the load and stores that we express in the source code, corresponding to the hardware locking protocol. It's true that Port is a memory-mapped object, so it too would be volatile, but we also need Temp to be volatile.

This high-level coding approach will work, and is simple enough that MCIs might not be needed. However, what really argues against it is that the correct sequence of emitted code requires the optimizer to remove all the other cruft that the code generator would otherwise include. (The gcc code generator used by the GNAT compiler generates initially poor code, by design, relying on the optimizer to clean it up.) In other words, we've told the optimizer not to change or add loads and stores for Temp, but without the optimizer enabled the code generator generates other code that gets in the way. That's OK in itself, as far as procedure Lock is concerned, but if the optimizer is sufficiently enabled we cannot debug the rest of the code. Using MCIs avoids these issues. The point, though, is that not all volatile objects are memory mapped.

So far we've been illustrating volatility with scalar objects, such as Lock.Temp above. What about objects of array and record types? (There are other "composite" types in Ada but they are not pertinent here.)

When aspect Volatile is applied to a record type or an object of such a type, all the record components are automatically volatile too.

For an array type (but not a record type), a related aspect Volatile_Components declares that the components of the array type — but not the array type itself — are volatile. However, if the Volatile aspect is specified, then the Volatile_Components aspect is automatically applied too, and vice versa. Thus components of array types are covered automatically.

If an object (of an array type or record type) is marked volatile then so are all of its sub-components, even if the type itself is not marked volatile.

Therefore aspects Volatile and Volatile_Components are nearly equivalent. In fact, Volatile_Components is superfluous. The language provides the Volatile_Components aspect only to give symmetry with the Atomic_Components and Independent_Components aspects. You can simply apply Volatile and be done with it.

Finally, note that applying aspect Volatile does not implicitly apply Independent, although

you can specify it explicitly if need be.

78.3 Aspect Atomic

Consider the GPIO pin configuration lock we've mentioned a few times now, that freezes the configuration of a given pin on a given GPIO port. The register, named LCKR for "lock register", occupies 32-bits, but only uses 17 total bits (currently). The low-order 16 bits, [0:15], represent the 16 GPIO pins on the given port. Bit #16 is the lock bit. That bit is the first bit in the upper half of the entire word. To freeze the configuration of a given pin in [0:15], the lock bit must be set at the same time as the bit to be frozen. In other words, the lower half and the upper half of the 32-bit word representing the register must be written together, at the same time. That way, accidental (un)freezing is unlikely to occur, because the most efficient, hence typical way for the generated code to access individual bits is for the compiler to load or store just the single byte that contains the bit or bits in question.

This indivisibility effect can be specified via aspect Atomic. As a result, all reads and updates of such an object as a whole are indivisible. In practice that means that the entire object is accessed with one load or store instruction. For a 16-bit object, all 16-bits are loaded and stored at once. For a 32-bit object, all 32-bits at once, and so on. The upper limit is the size of the largest machine scalar that the processor can manipulate with one instruction, as defined by the target processor. The typical lower bound is 8, for a byte-addressable machine.

Therefore, within the record type representing a GPIO port, we include the lock register component and apply the aspect Atomic:

```
type GPIO_Port is limited record
  ...
  LCKR : UInt32 with Atomic;
  ...
end record with
  ...
  Size => 16#400# * 8;
```

Hence loads and stores to the LCKR component will be done atomically, otherwise the compiler will let us know that it is impossible. That's all we need to do for the lock register to be read and updated atomically.

You should understand that only accesses to the whole, entire object are atomic. In the case of the lock register, the entire object is a record component, but that causes no problems here.

There is, however, something we must keep in mind when manipulating the values of atomic objects. For the lock register we're using a scalar type to represent the register, an unsigned 32-bit integer. There are no sub-components because scalar types don't have components, by definition. We simply use the bit-level operations to set and clear the individual bits. But we cannot set the bits — the lock bit and the bit for the I/O pin to freeze — one at a time because the locking protocol requires all the bits to be written at the same time, and only the entire 32-bit load and stores are atomic. Likewise, if instead of a scalar we used a record type or an array type to represent the bits in the lock register, we could not write individual record or array components one at a time, for the same reason we could not write individual bits using the unsigned scalar. The Atomic aspect only applies to loads and stores of the entire register.

Therefore, to update or read individual parts of an atomic object we must use a coding idiom in which we explicitly read or write the entire object to get to the parts. For example, to read an individual record component, we'd first read the entire record object into a temporary variable, and then access the component of that temporary variable. Likewise, to update one or more individual components, we'd first read the record object into a temporary variable, update the component or components within that temporary, and then write the

temporary back to the mapped device object. This is known as the "read-modify-write" idiom. You'll see this idiom often, regardless of the programming language, because the hardware requirement is not unusual. Fortunately Ada defines another aspect that makes the compiler do this for us. We'll describe it in the next section.

Finally, there are issues to consider regarding the other aspects described in this section.

If you think about atomic behavior in the context of machine instructions, loading and storing from/to memory atomically can only be performed for quantities that are independently addressable. Consequently, all atomic objects are considered to be specified as independently addressable too. Aspect specifications and representation items cannot change that fact. You can expect the compiler to reject any aspect or representation choice that would prevent this from being true.

Likewise, atomic accesses only make sense on actual memory locations, not registers. Therefore all atomic objects are volatile objects too, automatically.

However, unlike volatile objects, the components of an atomic object are not automatically atomic themselves. You'd have to mark these types or objects explicitly, using aspect `Atomic_Components`. Unlike `Volatile_Components`, aspect `Atomic_Components` is thus useful.

As is usual with Ada programming, you can rely on the compiler to inform you of problems. The compiler will reject an attempt to specify `Atomic` or `Atomic_Components` for an object or type if the implementation cannot support the indivisible and independent reads and updates required.

78.4 Aspect `Full_Access_Only`

Many devices have single-bit flags in the hardware that are not allocated to distinct bytes. They're packed into bytes and words shared with other flags. It isn't just individual bits either. Multi-bit fields that are smaller than a byte, e.g., two 4-bit quantities packed into a byte, are common. We saw that with the GPIO alternate functions codes earlier.

Ordinarily in Ada we represent such composite hardware interfaces using a record type. (Sometimes an array type makes more sense. That doesn't change anything here.) Compared to using bit-patterns, and the resulting bit shifting and masking in the source code, a record type representation and the resulting "dot notation" for accessing components is far more readable. It is also more robust because the compiler does all the work of retrieving these individual bits and bit-fields for us, doing any shifting and masking required in the generated code. The loads and stores are done by the compiler in whatever manner the compiler thinks most efficient.

When the hardware device requires atomic accesses to the memory mapped to such flags, we cannot let the compiler generate whatever width load and store accesses it thinks best. If full-word access is required, for example, then only loads and stores for full words can work. Yet aspect `Atomic` only guarantees that the entire object, in this case the record object, is loaded and stored indivisibly, via one instruction. The aspect doesn't apply to reads and updates to individual record components.

In the section on `Atomic` above, we mentioned that proper access to individual components of atomic types/objects can be achieved by a "read-modify-write" idiom. In this idiom, to read a component you first read into a temporary the entire enclosing atomic object. Then you read the individual component from that temporary variable. Likewise, to update an individual component, you start with the same approach but then update the component(s) within the temporary, then store the entire temporary back into the mapped atomic object. Applying aspect `Atomic` to the enclosing object ensures that reading and writing the temporary will be atomic, as required.

Using bit masks and bit patterns to access logical components as an alternative to a record type doesn't change the requirement for the idiom.

Consider the STM32F4 DMA device. The device contains a 32-bit stream configuration register that requires 32-bit reads and writes. We can map that register to an Ada record type like so:

```
type Stream_Config_Register is record
  -- ...
  Direction      : DMA_Data_Transfer_Direction;
  P_Flow_Controller : Boolean;
  TCI_Enabled     : Boolean; -- transfer complete
  HTI_Enabled     : Boolean; -- half-transfer complete
  TEI_Enabled     : Boolean; -- transfer error
  DMEI_Enabled    : Boolean; -- direct mode error
  Stream_Enabled  : Boolean;
end record
with Atomic, Size => 32;
```

The "confirming" size clause ensures we have declared the type correctly such that it will fit into 32-bits. There will also be a record representation clause to ensure the record components are located internally as required by the hardware. We don't show that part.

The aspect Atomic is applied to the entire record type, ensuring that the memory mapped to the hardware register is loaded and stored only as 32-bit quantities. In this example it isn't that we want the loads and stores to be indivisible. Rather, we want the generated machine instructions that load and store the object to use 32-bit word instructions, even if we are only reading or updating a component of the object. That's what the hardware requires for all accesses.

Next we'd use that type declaration to declare one of the components of an enclosing record type representing one entire DMA "stream":

```
type DMA_Stream is record
  CR : Stream_Config_Register;
  NDTR : Word; -- upper half must remain at reset value
  PAR : Address; -- peripheral address register
  M0AR : Address; -- memory 0 address register
  M1AR : Address; -- memory 1 address register
  FCR : FIFO_Control_Register;
end record
with Volatile, Size => 192; -- 24 bytes
```

Hence any individual DMA stream record object has a component named CR that represents the corresponding configuration register.

The DMA controllers have multiple streams per unit so we'd declare an array of DMA_Stream components. This array would then be part of another record type representing a DMA controller. Objects of the DMA_Controller type would be mapped to memory, thus mapping the stream configuration registers to memory.

Now, given all that, suppose we want to enable a stream on a given DMA controller. Using the read-modify-write idiom we would do it like so:

```
procedure Enable
  (Unit : in out DMA_Controller;
   Stream : DMA_Stream_Selector)
is
  Temp : Stream_Config_Register;
  -- these registers require 32-bit accesses, hence the temporary
begin
  Temp := Unit.Streams (Stream).CR; -- read entire CR register
  Temp.Stream_Enabled := True;
  Unit.Streams (Stream).CR := Temp; -- write entire CR register
end Enable;
```

That works, and of course the procedural interface presented to clients hides the details, as it should.

To be fair, the bit-pattern approach can express the idiom concisely, as long as you're careful. Here's the C code to enable and disable a selected stream:

```
#define DMA_SxCR_EN    ((uint32_t)0x00000001)

/* Enable the selected DMAy Streamx by setting EN bit */
DMAy_Streamx->CR |= DMA_SxCR_EN;

/* Disable the selected DMAy Streamx by clearing EN bit */
DMAy_Streamx->CR &= ~DMA_SxCR_EN;
```

The code reads and writes the entire CR register each time it is referenced so the requirement is met.

Nevertheless, the idiom is error-prone. We might forget to use it at all, or we might get it wrong in one of the very many places where we need to access individual components.

Fortunately, Ada provides a way to have the compiler implement the idiom for us, in the generated code. Aspect `Full_Access_Only` specifies that all reads of, or writes to, a component are performed by reading and/or writing all of the nearest enclosing full access object. Hence we add this aspect to the declaration of `Stream_Config_Register` like so:

```
type Stream_Config_Register is record
  -- ...
  Direction      : DMA_Data_Transfer_Direction;
  P_Flow_Controller : Boolean;
  TCI_Enabled    : Boolean; -- transfer complete interrupt
  HTI_Enabled    : Boolean; -- half-transfer complete
  TEI_Enabled    : Boolean; -- transfer error interrupt
  DMEI_Enabled   : Boolean; -- direct mode error interrupt
  Stream_Enabled : Boolean;
end record
with Atomic, Full_Access_Only, Size => 32;
```

Everything else in the declaration remains unchanged.

Note that `Full_Access_Only` can only be applied to `Volatile` types or objects. `Atomic` types are automatically `Volatile` too, so either one is allowed. You'd need one of those aspects anyway because `Full_Access_Only` just specifies the accessing instruction requirements for the generated code when accessing components.

The big benefit comes in the source code accessing the components. Procedure `Enable` is now merely:

```
procedure Enable
  (Unit   : in out DMA_Controller;
   Stream : DMA_Stream_Selector)
is
begin
  Unit.Streams (Stream).CR.Stream_Enabled := True;
end Enable;
```

This code works because the compiler implements the read-modify-write idiom for us in the generated code.

The aspect `Full_Access_Only` is new in Ada 2022, and is based on an implementation-defined aspect that GNAT first defined named `Volatile_Full_Access`. You'll see that GNAT aspect throughout the Arm device drivers in the Ada Drivers Library, available here: https://github.com/AdaCore/Ada_Drivers_Library. Those drivers were the motivation for the GNAT aspect.

Unlike the other aspects above, there is no pragma corresponding to the aspect `Full_Access_Only` defined by Ada 2022. (There is such a pragma for the GNAT-specific version named `Volatile_Full_Access`, as well as an aspect.)

HANDLING INTERRUPTS

79.1 Background

Embedded systems developers offload functionality from the application processor onto external devices whenever possible. These external devices may be on the same "chip" as the central processor (e.g., within a System-on-Chip) or they may just be on the same board, but the point here is that they are not the processor executing the application. Offloading work to these other devices enables us to get more functionality implemented in a target platform that is usually very limited in resources. If the processor has to implement everything we might miss deadlines or perhaps not fit into the available code space. And, of course, some specialized functionality may simply require an external device, such as a sensor.

For a simple example, a motor encoder is a device attached to a motor shaft that can be used to count the number of full or partial rotations that the shaft has completed. When the shaft is rotating quickly, the application would need to interact with the encoder frequently to get an up-to-date count, representing a non-trivial load on the application processor. There are ways to reduce that load, which we discuss shortly, but by far the simplest and most efficient approach is to do it all in hardware: use a timer device driven directly by the encoder. The timer is connected to the encoder such that the encoder signals act like an external clock driving the timer's internal counter. All the application processor must do to get the encoder count is query the timer's counter. The timer is almost certainly memory-mapped, so querying the timer amounts to a memory access.

In some cases, we even offload communication with these external devices onto other external devices. For example, the I2C⁴²⁵ (Inter-Integrated Circuit) protocol is a popular two-wire serial protocol for communicating between low-level hardware devices. Individual bits of the data are sent by driving the data line high and low in time with the clock signal on the other line. The protocol has been around for a long time and many embedded devices use it to communicate. We could have the application drive the data line for each individual bit in the protocol. Known as "bit-banging," that would be a significant load on the processor when the overall traffic volume is non-trivial. Fortunately, there are dedicated devices — I2C transceivers — that will implement the protocol for us. To send application data to another device using the I2C protocol, we just give the transceiver the data and destination address. The rest is done in the transceiver hardware. Receiving data is of course also possible. I2C transceivers are ubiquitous because the protocol is so common among device implementations. A USART⁴²⁶ / UART⁴²⁷ is a similar example.

Having offloaded some of the work, the application must have some way to interact with the device in order to know what is happening. Maybe the application has requested the external device perform some service — an analog-to-digital conversion, say — and must know when that function has completed. Maybe a communications device is receiving incoming data for the application to process. Or maybe that communications device has completed sending outgoing data and is ready for more to send.

⁴²⁵ <https://en.wikipedia.org/wiki/I%C2%B2C>

⁴²⁶ https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

⁴²⁷ https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Ultimately, interaction with the external device will be either synchronous or asynchronous, and has system-level design implications.

For synchronous interaction, the application periodically queries the device, typically a status flag or function on the device. Known as "polling," this approach is simple to implement but wastes cycles when the external device has not yet completed the request. After all, the point of offloading the work is to allow the application processor to execute other functionality. Polling negates that benefit. On the other hand, if the expected time to completion is extremely short, polling can be sufficiently efficient to make sense.

Usually, there's enough time involved so that polling is undesirable. The external environment takes time to respond and change state. Maybe a sensor has been designed to wait passively for something to happen in the external world, and only on the infrequent occurrence of that event should the application be notified. Perhaps a switch is to be toggled in certain circumstances, or an intruder detected. In this case, nothing happens for extended intervals.

As a consequence of all this, there's a very good chance that the internal processor should not poll these external devices.

Before we discuss the asynchronous alternative, there's another issue to consider. However the notification from the external device is implemented, a very quick response from the internal processor may be required. Think back to that serial port with a USART again. The USART is responsible for composing the arriving characters (or bytes) from their individual incoming bits on the receiving line. When all the bits for a single character have arrived, what happens next depends on the software design. In the simplest case, the internal processor copies the single character from the USART to an internal buffer and then goes back to doing something else while the next full character arrives in the USART. The response to the USART must be fairly quick because the next incoming character's bits are arriving. The internal processor must get the current character before it is overwritten by the next arriving character, otherwise we'll lose data. So we can say that the response to the notification from the external device must often be very quick.

Now, ideally in the USART case, we would further offload the work from the internal processor. Instead of having the processor copy each arriving character from the USART into an application buffer, we would have another external hardware device — a [direct memory access \(DMA\)](https://en.wikipedia.org/wiki/Direct_memory_access)⁴²⁸ device — copy each arriving character from the USART to the buffer. A DMA device copies data from one location to another, in this case from the address of the USART's one-character memory-mapped register to the address of the application buffer in memory. The copy is performed by the DMA hardware so it is extremely fast and costs the main processor no cycles. But even with this approach, we need to notify the application that a complete message is ready for processing. We might need to do that quickly so that enough time remains for the application to process the message content prior to the arrival of the next message.

Therefore, the general requirement is for an external device to be able to asynchronously notify the internal processor, and for the notification to be implemented in such a way that the beginning of the response can be sufficiently and predictably quick.

Fortunately, computers already have such a mechanism: interrupts. The details vary considerably with the hardware architecture, but the overall idea is independent of the [ISA](https://en.wikipedia.org/wiki/ISA)⁴²⁹: an external event can trigger a response from the processor by becoming "active." The current state of the application is temporarily stored, and then an interrupt response routine, known as an "interrupt handler" is executed. Upon completion of the handler, the original state of the application is restored and the application continues execution. The time between the interrupt becoming active and the start of the responding handler execution is known as the "interrupt latency."

Hardware interrupts typically have priorities assigned, depending on the hardware. These priorities are applied when multiple interrupts are triggered at the same time, to define

⁴²⁸ https://en.wikipedia.org/wiki/Direct_memory_access

⁴²⁹ https://en.wikipedia.org/wiki/Instruction_set_architecture

the order in which the interrupts are presented and the handlers invoked. The canonical model is that only higher-priority interrupts can preempt handlers executing in response to interrupts with lower or equal priority.

Ada defines a model for hardware interrupts and interrupt handling that closely adheres to the conceptual model described above. If you have experience with interrupt handling, you will recognize them in the Ada model. One very important point to make about the Ada facilities is that they are highly portable, so they don't require extensive changes when moving to a new target computer. Part of that portability is due to the language-defined model.

Before we go into the Ada facility details, there's a final point. Sometimes we *do* want the application to wait for the external device. When would that be the case? To answer that, we need to introduce another term. The act of saving and restoring the state of the interrupted application software is known as "interrupt context switching." If the time for the device to complete the application request is approximately that of the context switching, the application might as well wait for the device after issuing the request.

Another reason to consider polling is that the architectural complexity of interrupt handling is greater than that of polling. If your system has some number of devices to control and polling them would be fast enough for the application to meet requirements, it is simpler to do so. But that will likely only work for a few devices, or at least a few that have short response time requirements.

The application code can wait for the device by simply entering a loop, exiting only when some external device status flag indicates completion of the function. The loop itself, in its simplest form, would contain only the test for exiting. As mentioned earlier, polling in a tight loop like this only makes sense for very fast device interactions. That's not the usual situation though, so polling should not be your default design assumption. Besides, active polling consumes power. On an embedded platform, conserving power is often important.

That loop polling the device will never exit if the device can fail to signal completion. Or maybe it might take too long in some odd case. If you don't want to be potentially stuck in the loop indefinitely, chewing up cycles and power, you can add an upper bound on the number of attempts, i.e., loop iterations. For example:

```

procedure Await_Data_Ready (This : in out Three_Axis_Gyroscope) is
  Max_Status_Attempts : constant := 10_000;
  -- This upper bound is arbitrary but must be sufficient for the
  -- slower gyro data rate options and higher clock rates. It need
  -- not be as small as possible, the point is not to hang forever.
begin
  Polling: for K in 1 .. Max_Status_Attempts loop
    if Data_Status (This).ZYX_Available then
      return;
    end if;
  end loop Polling;
  raise Gyro_Failure;
end Await_Data_Ready;

```

In the above, Data_Status is a function that returns a record object containing Boolean flags. The if-statement queries one of those flags. Thus the loop either detects the desired device status or raises an exception after the maximum number of attempts have been made. In this version, the maximum is a known upper bound so a local constant will suffice. The maximum could be passed as a parameter instead, or declared in a global "configuration" package containing such constants.

Presumably, the upper bound on the attempts is either specified by the device documentation or empirically determined. Sometimes, however, the documentation will instead specify a maximum possible response time, for instance 30 milliseconds. Any time beyond that maximum indicates a device failure.

In the code above, the number of iterations indirectly defines the amount of elapsed time

the caller waits. That time varies with the target's system clock and the generated instructions' required clock cycles, hence the approach is not portable. Alternatively, we can work in terms of actual time, which will be portable across all targets with a sufficiently precise clock.

You can use the facilities in package `Ada.Real_Time` to work with time values. That package defines a type `Time_Span` representing time intervals, useful for expressing relative values such as elapsed time. There is also type `Time` representing an absolute value on the timeline. A function `Clock` returns a value of type `Time` representing "now," along with overloaded addition and subtraction operators taking `Time` and `Time_Span` parameters. The package also provides operators for comparing `Time` values. (The value returned by `Clock` is monotonically increasing so you don't need to handle time zone jumps and other such things, unlike the function provided by `Ada.Calendar`.)

If the timeout is not context-specific then we'd use a constant as we did above, otherwise we'd allow the caller to specify the timeout. For example, here's a polling routine included with the DMA device driver we've mentioned a few times now. Some device-specific parts have been removed to keep the example simple. The appropriate timeout varies, so it is a parameter to the call:

```
procedure Poll_For_Completion
  (This      : in out DMA_Controller;
   Stream    : DMA_Stream_Selector;
   Timeout   : Time_Span;
   Result    : out DMA_Error_Code)
is
  Deadline : constant Time := Clock + Timeout;
begin
  Result := DMA_No_Error; -- initially
  Polling : loop
    exit Polling when Status (This, Stream, Transfer_Complete_Indicated);
    if Clock >= Deadline then
      Result := DMA_Timeout_Error;
      return;
    end if;
  end loop Polling;
  Clear_Status (This, Stream, Transfer_Complete_Indicated);
end Poll_For_Completion;
```

In this approach, we compute the deadline as a point on the timeline by adding the value returned from the `Clock` function (i.e., "now") to the time interval specified by the parameter. Then, within the loop, we compare the value of the `Clock` to that deadline.

Finally, with another design approach we can reduce the processor cycles "wasted" when the polled device is not yet ready. Specifically, in the polling loop, when the device has not yet completed the requested function, we can temporarily relinquish the processor so that other tasks within the application can execute. That isn't perfect because we're still checking the device status even though we cannot exit the loop. And it requires other tasks to exist in your design, although that's probably a good idea for other reasons (e.g., logical threads having different, non-harmonic periods). This approach would look like this (an incomplete example):

```
procedure Poll_With_Delay is
  Next_Release : Time;
  Period       : constant Time_Span := Milliseconds (30); -- let's say
begin
  Next_Release := Clock;
  loop
    exit when Status (...);
    Next_Release := Next_Release + Period;
    delay until Next_Release;
```

(continues on next page)

(continued from previous page)

```
end loop;  
end Poll_With_Delay;
```

The code above will check the status of some device every 30 milliseconds (an arbitrary period just for illustration) until the Status function result allows the loop to exit. If the device "hangs" the loop is never exited, but as you saw there are ways to address that possibility. When the code does not exit the loop, the next point on the timeline is computed and the task executing the code then suspends, allowing the other tasks in the application to execute. Eventually, the next release point is reached and so the task becomes ready to execute again (and will, subject to priorities).

But how long should the polling task suspend when awaiting the device? We need to suspend long enough for the other tasks to get something done, but not so long that the device isn't handled fast enough. Finding the right balance is often not simple, and is further complicated by the "task switching" time. That's the time it takes to switch the execution context from one task to another, in this case in response to the "delay until" statement suspending the polling task. And it must be considered in both directions: when the delay expires we'll eventually switch back to the polling task.

As you can see, polling is easily expressed but has potentially significant drawbacks and architectural ramifications so it should be avoided as a default approach.

Now let's explore the Ada interrupt facilities.

79.2 Language-Defined Interrupt Model

The Ada language standard defines a model for hardware interrupts, as well as language-defined mechanisms for handling interrupts consistent with that model. The model is defined in Annex C, the "Systems Programming" annex, section 3 "Interrupt Support." The following is the text of that section with only a few simplifications and elisions.

- Interrupts are said to occur. An occurrence of an interrupt is separable into generation and delivery.
 - Generation of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program.
 - Delivery is the action that invokes part of the program as response to the interrupt occurrence.
- Between generation and delivery, the interrupt occurrence is pending.
- Some or all interrupts may be blocked. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.
- Certain interrupts are reserved. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other RTL-defined means. The set of reserved interrupts is determined by the hardware and run-time library (RTL).
- Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be attached to that interrupt. The execution of that program unit, the interrupt handler, is invoked upon delivery of the interrupt occurrence.
- While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt.
- The corresponding interrupt is blocked while the handler executes. While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is determined by the hardware and the RTL.

- Each interrupt has a default treatment which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is defined by the RTL.
- An exception propagated from a handler that is invoked by an interrupt has no effect. In particular, it is not propagated out of the handler, in the same way that exceptions do not propagate outside of task bodies.
- If the `Ceiling_Locking` policy is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. ("Protected object" is abbreviated as "PO" for convenience).
- If the hardware or the underlying system holds pending interrupt occurrences, the RTL must provide for later delivery of these occurrences to the program.

(The above is not everything in the model but we can ignore the rest in this introduction.)

Because interrupt occurrences are generated by the hardware and delivered by the underlying system software (run-time library or real-time operating system), the application code is mainly responsible for responding to occurrences. Of course, the application must first configure the relevant external devices so that they generate the expected interrupts.

The actual response is application-specific but is also hardware-specific. The latter often (but not always) requires clearing the interrupt status within the generating device so that the same occurrence is not delivered again.

Furthermore, the standard model requires the underlying software to block further occurrences while the handler executes, and only allow preemption by higher-priority interrupt occurrences (if any). The application handlers are not responsible for these semantics either. As you will see, the choice of program unit used for expressing handlers makes this all very convenient for the developer.

As a consequence, in terms of the response, the application developer must write the specific handlers and attach those handlers to the corresponding interrupts. Attaching the handlers is implemented in the underlying system software, and it is this same underlying software that delivers the occurrences.

We will now explore the Ada facilities in detail. At the end of this chapter we will explore some common idioms using these mechanisms, especially with regard to the handlers' interaction with the rest of the application.

79.3 Interrupt Handlers

Interrupt handling is, by definition, asynchronous: some event occurs that causes the processor to suspend the application, respond to the event, and then resume application execution.

Because these events are asynchronous, the actions performed by the interrupt handler and the application are subject to the same sorts of race conditions as multiple tasks acting on shared data.

For example, a "reader" task may be in the act of reading (copying) the value of some shared variable, only to be preempted by a "writer" task that updates the value of the variable. In that case, when the "reader" task resumes execution, it will finish the read operation but will, as a result, have a value that is partly from the old value and partly from the new value. The effect is unpredictable. An interrupt handler can have the same effect on shared data as the preempting "writer" task that interrupts the "reader" task. This problem is possible for shared data of any type that is not atomically read or written. You can think of large record objects if that helps, but it even applies to some scalars.

That scenario applies even if no explicit tasks are declared in the application. That's because an implicit "environment task" is executing the main subprogram. In that case, the

main subprogram is the entire application, but more typically some non-null application code is actively executing in one or more tasks.

But it's not just a matter of tasks. We said that interrupts usually have priorities. Typically that means a higher-priority interrupt will preempt the execution of the handler for a lower-priority interrupt. It's the same issue.

Furthermore, the fact that an interrupt has occurred needs to be communicated to the application, for example to say that updated data are available, perhaps a sensor reading or characters from a serial port. As we said above, we usually don't want to poll for that fact, so the application must be able to suspend until the event has occurred. Often we'll have a dedicated task within the application that suspends, rather than the entire application, but that's an application detail.

Ada's protected objects address all these asynchronous issues. Shared data declared within a protected object can be accessed only via protected procedures or protected entries, both of which execute with mutually exclusive access. Hence no race conditions are possible.

Here is an extremely simple, but realistic, example of a PO. This is not an interrupt handler example — we'll get to that — but it does show a shared variable and a protected procedure that executes with mutually exclusive access no matter how many tasks concurrently call it. The PO provides unique serial numbers.

```
protected Serial_Number is
  procedure Get_Next (Number : out Positive);
private
  Value : Positive := 1;
end Serial_Number;

protected body Serial_Number is

  procedure Get_Next (Number : out Positive) is
  begin
    Number := Value;
    Value := Value + 1;
  end Get_Next;

end Serial_Number;
```

Imagine there are multiple assembly lines creating devices of various sorts. Each device gets a unique serial number. These assembly lines run concurrently, so the calls to `Get_Next` occur concurrently. Without mutually exclusive access to the `Value` variable, multiple devices could get the same serial number.

Protected entries can suspend a caller until some condition is true; in this case, the fact that an interrupt has occurred and been handled. (As we will see, a protected entry is not the only way to synchronize with an accessing task, but it is the most robust and general.)

Here's an example of a PO with a protected entry:

```
protected type Persistent_Signal is
  entry Wait;
  procedure Send;
private
  Signal_Arrived : Boolean := False;
end Persistent_Signal;

protected body Persistent_Signal is

  entry Wait when Signal_Arrived is
  begin
```

(continues on next page)

(continued from previous page)

```
    Signal_Arrived := False;
end Wait;

procedure Send is
begin
    Signal_Arrived := True;
end Send;

end Persistent_Signal;
```

This is a PO providing a "Persistent Signal" abstraction. It allows a task to wait for a "signal" from another task. The signal is not lost if the receiving task is not already waiting, hence the term "persistent." Specifically, if `Signal_Arrived` is `False`, a caller to `Wait` will be suspended until `Signal_Arrived` becomes `True`. A caller to `Send` sets `Signal_Arrived` to `True`. If a caller to `Wait` was already present, suspended, it will be allowed to continue execution. If no caller was waiting, eventually some caller will arrive, find `Signal_Arrived` `True`, and will be allowed to continue. In either case, the `Signal_Arrived` flag will be set back to `False` before the `Wait` caller is released. Protected objects can have a priority assigned, similar to tasks, so they are integrated into the global priority semantics including interrupt priorities.

Therefore, in Ada an interrupt handler is a protected procedure declared within some protected object (PO). A given PO may handle more than one interrupt, and if so, may use one or more protected procedures to do so.

Interrupts can be attached to a protected procedure handler using a mechanism we'll discuss shortly. When the corresponding interrupt occurs, the attached handler is invoked. Any exceptions propagated by the handler's execution are ignored and do not go past the procedure.

While the protected procedure handler executes, the corresponding interrupt is blocked. As a consequence, another occurrence of that same interrupt will not preempt the handler's execution. However, if the hardware does not allow interrupts to be blocked, no blocking occurs and a subsequent occurrence would preempt the current execution of the handler. In that case, your handlers must be written with that possibility in mind. Most targets do block interrupts so we will assume that behavior in the following descriptions.

The standard mutually exclusive access provided to the execution of protected procedures and entries is enforced whether the "call" originates in hardware, via an interrupt, or in the application software, via some task. While any protected action in the PO executes, the corresponding interrupt is blocked, such that another occurrence will not preempt the execution of that actions' procedure or entry body execution in the PO.

On some processors blocked interrupts are lost, they do not persist. However, if the hardware can deliver an interrupt that had been blocked, the Systems Programming Annex requires the handler to be invoked again later, subject to the PO semantics described above.

The default treatment for a given interrupt depends on the RTL implementation. The default may be to jump immediately to system-defined handler that merely loops forever, thereby "hanging" the system and preventing any further execution of the application. On a bare-board target that would be a very common approach. Alternatively the default could be to ignore the interrupt entirely.

As mentioned earlier, some interrupts may be reserved, meaning that the application cannot install a replacement handler. For instance, most bare-board systems include a clock that is driven by a dedicated interrupt. The application cannot (or at least should not) override the interrupt handler for that interrupt. The determination of which interrupts are reserved is RTL-defined. Attempting to attach a user-defined handler for a reserved interrupt raises `Program_Error`, and the existing treatment is unchanged.

79.4 Interrupt Management

Ada defines a standard package that provides a primary type for identifying individual interrupts, as well as subprograms that take a parameter of that type in order to manage the system's interrupts and handlers. The package is named `Ada.Interrupts`, appropriately.

The primary type in that package is named `Interrupt_Id` and is an compiler-defined discrete type, meaning that it is either an integer type (signed or not) or an enumeration type. That representation is guaranteed so you can be sure that `Interrupt_Id` can be used, for example, as the index for an array type.

Package `Ada.Interrupts` provides functions to query whether a given interrupt is reserved, or if an interrupt has a handler attached. Procedures are defined to allow the application to attach and detach handlers, among other things. These procedures allow the application to dynamically manage interrupts. For example, when a new external device is added, perhaps as a "hot spare" replacing a damaged device, or when a new external device is simply connected to the target, the application can arrange to handle the new interrupts without having to recompile the application or restart application execution.

However, typically you will not use these procedures or functions to manage interrupts. In part that's because the architecture is usually static, i.e., the handlers are set up once and then never changed. In that case you won't need to query whether a given exception is reserved at run-time, or to check whether a handler is attached. You'd know that already, as part of the system architecture choices. For the same reasons, another mechanism for attaching handlers is more commonly used, and will be explained in that section. The package's type `Interrupt_Id`, however, will be used extensively.

A child package `Ada.Interrupts.Names` defines a target-dependent set of constants providing meaningful names for the `Interrupt_Id` values the target supports. Both the number of constants and their names are defined by the compiler, reflecting the variations in hardware available. This package and the enclosed constants are used all the time. For the sake of illustration, here is part of the package declaration for a Cortex M4F microcontroller supported by GNAT:

```
package Ada.Interrupts.Names is
  Sys_Tick_Interrupt      : constant Interrupt_Id := 1;
  ...
  EXTI0_Interrupt        : constant Interrupt_Id := 8;
  ...
  DMA1_Stream0_Interrupt : constant Interrupt_Id := 13;
  ...
  HASH_RNG_Interrupt     : constant Interrupt_Id := 80;
  ...
end Ada.Interrupts.Names;
```

Notice `HASH_RNG_Interrupt`, the name for `Interrupt_Id` value 80 on this target. That is the interrupt that the on-chip random number generator hardware uses to signal that a new value is available. We will use this interrupt in an example at the end of this chapter.

The representation chosen by the compiler for `Interrupt_Id` is very likely an integer, as in the above package, so the child package provides readable names for the numeric values. If `Interrupt_Id` is represented as an enumeration type the enumerals are probably sufficiently readable, but the child package must be provided by the vendor nonetheless.

79.5 Associating Handlers With Interrupts

As we mentioned above, the Ada standard provides two ways to attach handlers to interrupts. One is procedural, described earlier. The other mechanism is automatic, achieved during elaboration of the protected object enclosing the handler procedure. The behavior is not unlike the activation of tasks: declared tasks are activated automatically as a result

of their elaboration, whereas dynamically allocated tasks are activated as a result of their allocations.

We will focus exclusively on the automatic, elaboration-driven attachment model because that is the more common usage, and as a result, that is what GNAT supports on bare-board targets. It is also the mechanism that the standard Ravenscar and Jorvik profiles require. Our examples are consistent with those targets.

In the elaboration-based attachment model, we specify the interrupt to be attached to a given protected procedure within a protected object. This interrupt specification occurs within the enclosing protected object declaration. (Details in a moment.) When the enclosing PO is elaborated, the run-time library installs that procedure as the handler for that interrupt. A given PO may contain one or more interrupt handler procedures, as well as any other protected subprograms and entries.

In particular, we can associate an interrupt with a protected procedure by applying the aspect `Attach_Handler` to that procedure as part of its declaration, with the `Interrupt_Id` value as the aspect parameter. The association can also be achieved via a pragma with the same name as the aspect. Strictly speaking, the pragma `Attach_Handler` is obsolescent, but that just means that there is a newer way to make the association (i.e., the aspect). The pragma is not illegal and will remain supported. Because the pragma existed in a version of Ada prior to aspects you will see a lot of existing code using the pragma. You should become familiar with it. There's no language-driven reason to change the source code to use the aspect. New code should arguably use the aspect, but there's no technical reason to prefer one over the other.

Here is an example of a protected object with one protected procedure interrupt handler. It uses the `Attach_Handler` aspect to tie a random number generator interrupt to the `RNG_Controller.Interrupt_Handler` procedure:

```
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private
  Last_Sample      : UInt32 := 0;
  Buffer            : Ring_Buffer;
  Data_Available   : Boolean := False;

  procedure Interrupt_Handler with
    Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;
end RNG_Controller;
```

That's all that the developer must do to install the handler. The compiler and run-time library do the rest, automatically.

The local variables are declared in the private part, as required by the language, because they are shared data meant to be protected from race conditions. Therefore, the only compile-time access possible is via visible subprograms and entries declared in the visible part. Those subprograms and entries execute with mutually exclusive access so no race conditions are possible, as guaranteed by the language.

Note that procedure `Interrupt_Handler` is declared in the private part of `RNG_Controller`, rather than the visible part. That location is purely a matter of choice (unlike the variables), but there is a good reason to hide it: application software can call an interrupt handler procedure too. If you don't ever intend for that to happen, have the compiler enforce your intent. An alert code reader will then recognize that clients cannot call that procedure. If, on the other hand, the handler is declared in the visible part, the reader must examine more of the code to determine whether there are any callers in the application code. Granted, a software call to an interrupt handler is rare, but not illegal, so you should state your intent in the code in an enforceable manner.

Be aware that the Ada compiler is allowed to place restrictions on protected procedure handlers. The compiler can restrict the content of the procedure body, for example, or it might forbid calls to the handler from the application software. The rationale is to allow direct invocation by the hardware, to minimize interrupt latency to the extent possible.

For completeness, here's the same RNG_Controller protected object using the pragma instead of the aspect to attach the interrupt to the handler procedure:

```
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private
  Last_Sample      : UInt32 := 0;
  Buffer            : Ring_Buffer;
  Data_Available   : Boolean := False;

  procedure Interrupt_Handler;
  pragma Attach_Handler (Interrupt_Handler,
                        Ada.Interrupts.Names.HASH_RNG_Interrupt;

end RNG_Controller;
```

As you can see, there isn't much difference. The aspect is somewhat more succinct. (The choice of where to declare the procedure remains the same.)

In this attachment model, protected declarations containing interrupt handlers must be declared at the library level. That means they must be declared in library packages. (Protected objects cannot be library units themselves, just as tasks cannot. They must be declared within some other unit.) Here is the full declaration for the RNG_Controller PO declared within a package — in this case within a package body:

```
with Ada.Interrupts.Names;
with Bounded_Ring_Buffers;

package body STM32.RNG.Interrupts is

  package UInt32_Buffers is new Bounded_Ring_Buffers (Content => UInt32);
  use UInt32_Buffers;

  protected RNG_Controller is
    ...
    entry Get_Random (Value : out UInt32);
  private
    Last_Sample      : UInt32 := 0;
    Samples          : Ring_Buffer (Upper_Bound => 9); -- arbitrary
    Data_Available   : Boolean := False;

    procedure Interrupt_Handler with
      Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;

  end RNG_Controller;

  ...

end STM32.RNG.Interrupts;
```

But note that we're talking about protected declarations, a technical term that encompasses not only protected types but also anonymously-typed protected objects. In the RNG_Controller example, the PO does not have an explicit type declared; it is anonymously-typed. (Task objects can also be anonymously-typed.) You don't have to

use a two-step process of first declaring the type and then an object of the type. If you only need one, no explicit type is required.

Although interrupt handler protected types must be declared at library level, the Ada model allows you to have an object of the type declared elsewhere, not necessarily at library level. However, note that the Ravenscar and Jorvik profiles require protected interrupt handler objects — anonymously-typed or not — to be declared at the library level too, for the sake of analysis. The profiles also require the elaboration-based attachment mechanism we have shown. For the sake of the widest applicability, and because with GNAT the most likely use-case involves either Ravenscar or Jorvik, we are following those restrictions in our examples.

79.6 Interrupt Priorities

Many (but not all) processors assign priorities to interrupts, with blocking and preemption among priorities of different levels, much like preemptive priority-based task semantics. Consequently, the priority semantics for interrupt handlers are as if a hardware "task," executing at an interrupt level priority, calls the protected procedure handler.

Interrupt handlers in Ada are protected procedures, which do not have priorities individually, but the enclosing protected object can be assigned a priority that will apply to the handler(s) when executing.

Therefore, protected objects can have priorities assigned using values of subtype `System.Interrupt_Priority`, which are high enough to require the blocking of one or more interrupts. The specific values among the priority subtypes are not standardized but the intent is that interrupt priorities are higher (more urgent) than non-interrupt priorities, as if they are declared like so in package `System`:

```
subtype Any_Priority is Integer range compiler-defined;

subtype Priority is Any_Priority
  range Any_Priority'First .. compiler-defined;

subtype Interrupt_Priority is Any_Priority
  range Priority'Last + 1 .. Any_Priority'Last;
```

For example, here are the subtype declarations in the GNAT compiler for an Arm Cortex M4 target:

```
subtype Any_Priority is Integer range 0 .. 255;
subtype Priority is Any_Priority range Any_Priority'First .. 240;
subtype Interrupt_Priority is Any_Priority range
  Priority'Last + 1 .. Any_Priority'Last;
```

Although the ranges are compiler-defined, when the Systems Programming Annex is implemented the range of `System.Interrupt_Priority` must include at least one value. Vendors are not required to have a distinct priority value in `Interrupt_Priority` for each hardware interrupt possible on a given target. On a bare-metal target, they probably will have a one-to-one correspondence, but might not in a target with an RTOS or host OS.

A PO containing an interrupt handler procedure must be given a priority within the `Interrupt_Priority` subtype's range. To do so, we apply the aspect `Interrupt_Priority` to the PO. Perhaps confusingly, the aspect and the value's required subtype have the same name.

```
with Ada.Interrupts.Names; use Ada.Interrupts.Names;
with System; use System;

package Gyro_Interrupts is
```

(continues on next page)

(continued from previous page)

```

protected Handler with
  Interrupt_Priority => Interrupt_Priority'Last
is
private
  procedure IRQ_Handler;
  pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
end Handler;

end Gyro_Interrupts;

```

The code above uses the highest (most urgent) interrupt priority value but some other value could be used instead, as long as it is in the `Interrupt_Priority` subtype's range. `Constraint_Error` is raised otherwise.

There is also an alternative pragma, now obsolescent, with the same name as the aspect and subtype. Here is an example:

```

with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Gyro_Interrupts is

  protected Handler is
    pragma Interrupt_Priority (245);
  private
    procedure IRQ_Handler;
    pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
  end Handler;

end Gyro_Interrupts;

```

In the above we set the interrupt priority to 245, presumably a value conformant with this specific target. You should be familiar with this pragma too, because there is some much existing code using it. New code should use the aspect, ideally.

If we don't specify the priority for some protected object containing an interrupt handler (using either the pragma or the aspect), the initial priority of protected objects of that type is compiler-defined, but within the range of the subtype `Interrupt_Priority`. Generally speaking, you should specify the priorities per those of the interrupts handled, assuming they have distinct values, so that you can reason concretely about the relative blocking behavior at run-time.

Note that the parameter specifying the priority is optional for the `Interrupt_Priority` pragma. When none is given, the effect is as if the value `Interrupt_Priority'Last` was specified.

```

with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Gyro_Interrupts is

  protected Handler is
    pragma Interrupt_Priority;
  private
    ...
  end Handler;

end Gyro_Interrupts;

```

No pragma parameter is given in the above, therefore `Gyro_Interrupts.Handler` executes at `Interrupt_Priority'Last` when invoked.

While an interrupt handler is executing, the corresponding interrupt is blocked. Therefore,

the same interrupt will not be delivered again while the handler is executing. Plus, the protected object semantics mean that no software caller is also concurrently executing within the protected object. So no data race conditions are possible. If the system does not support blocking, however, the interrupt is not blocked when the handler executes.

In addition, when interrupt priorities are involved, hardware blocking typically extends to interrupts of equal or lower priority.

You should understand that a higher-priority interrupt could preempt the execution of a lower-priority interrupt's handler. Handlers do not define "critical sections" in which the processor cannot be preempted at all (other than the case of the highest priority interrupt).

Preemption does not cause data races, usually, because the typical case is to have a given protected object handle only one interrupt. It follows that only that one interrupt handler has visibility to the protected data in any given protected object, therefore only that one handler can update it. Any preempting handler would be in a different protected object, hence the preempting handler could not possibly update the data in the preempted handler's PO. No data race condition is possible.

However, protected objects can contain handlers for more than one interrupt. In that case, depending on the priorities, the execution of a higher-priority handler could preempt the execution of a lower priority handler in that same PO. Because each handler in the PO can update the local protected data, these data are effectively shared among asynchronous writers. Data race conditions are, as a result, possible.

The solution to the case of multiple handlers in a single PO is to assign the PO a priority not less than the highest of the interrupt priorities for which it contains handlers. That's known as the "ceiling priority" and works the same as when applying the ceiling for the priorities of caller tasks in the software. Then, whenever any interrupt handled by that PO is delivered, the handler executes at the ceiling priority, not necessarily the priority of the specific interrupt handled. All interrupts at a priority equal or lower than the PO priority are blocked, so no preemption by another handler within that same PO is possible. As a result, a handler for a higher priority interrupt must be in a different PO. If that higher priority handler is invoked, it can indeed preempt the execution of the handler for the lower priority interrupt in another PO. But because these two handlers will not be in the same PO, they will not share the data, so again no race condition is possible.

Note also that software callers will execute at the PO priority as well, so their priority may be increased during that execution. As you can see, the Ceiling Priority Protocol integrates application-level priorities, for tasks and protected objects, with interrupt-level priorities for interrupt handlers.

The Ceiling Locking Protocol is requested by specifying the `Ceiling_Locking` policy (see ARM D.3) to the pragma `Locking_Policy`. Both Ravenscar and Jorvik do so, automatically.

79.7 Common Design Idioms

In this section we explore some of the common idioms used when writing interrupt handlers in Ada.

79.7.1 Parameterizing Handlers

Suppose we have more than one instance of a kind of device. For example, multiple DMA controllers are often available on a System-on-Chip such as an Arm microcontroller. We can simplify our code by defining a device driver **type**, with one object of the type per supported hardware device. This is the same abstract data type (ADT) approach we'd take for software objects in application code, and in general for device drivers when multiple hardware instances are available.

We can also apply the ADT approach to interrupt handlers when we have multiple devices of a given kind that can generate interrupts. In this case, the type will be fully implemented

as a protected type containing at least one interrupt handling procedure, with or without additional protected procedures or entries.

As is the case with abstract data types in general, we can tailor each object with discriminants defined with the type, in order to "parameterize" the type and thus allow distinct objects to have different characteristics. For example, we might define a bounded buffer ADT with a discriminant specifying the upper bound, so that distinct objects of the single type could have different bounds. In the case of hardware device instances, one of these parameters will often specify the device being driven, but we can also specify other device-specific characteristics. In particular, for interrupt handler types both the interrupt to handle and the interrupt priority can be discriminants. That's possible because the aspects/pragmas do not require their values to be specified via literals, unlike what was done in the `RNG_Controller` example above.

For example, here is the declaration for an interrupt handler ADT named `DMA_Interrupt_Controller`. This type manages the interrupts for a given DMA device, known as a `DMA_Controller`. Type `DMA_Controller` is itself an abstract data type, declared elsewhere.

```
protected type DMA_Interrupt_Controller
  (Controller : not null access DMA_Controller;
   Stream     : DMA_Stream_Selector;
   IRQ        : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is

  procedure Start_Transfer
    (Source      : Address;
     Destination : Address;
     Data_Count  : UInt16);

  procedure Abort_Transfer (Result : out DMA_Error_Code);

  procedure Clear_Transfer_State;

  function Buffer_Error return Boolean;

  entry Wait_For_Completion (Status : out DMA_Error_Code);

private

  procedure Interrupt_Handler with Attach_Handler => IRQ;

  No_Transfer_In_Progress : Boolean := True;
  Last_Status              : DMA_Error_Code := DMA_No_Error;
  Had_Buffer_Error         : Boolean := False;

end DMA_Interrupt_Controller;
```

In the above, the `Controller` discriminant provides an access value designating the specific `DMA_Controller` device instance to be managed. Each DMA device supports multiple independent conversion "streams" so the `Stream` discriminant specifies that characteristic. The `IRQ` and `IRQ_Priority` discriminants specify the handler values for that specific device and stream. These discriminant values are then used in the `Interrupt_Priority` pragma and the `Attach_Handler` aspect in the private part. ("IRQ" is a command handler name across programming languages, and is an abbreviation for "interrupt request.")

Here then are the declarations for two instances of the interrupt handler type:

```
DMA2_Stream0 : DMA_Interrupt_Controller
  (Controller => DMA_2'Access,
   Stream    => Stream_0,
   IRQ       => DMA2_Stream0_Interrupt,
   IRQ_Priority => Interrupt_Priority'Last);

DMA2_Stream5 : DMA_Interrupt_Controller
  (Controller => DMA_2'Access,
   Stream    => Stream_5,
   IRQ       => DMA2_Stream5_Interrupt,
   IRQ_Priority => Interrupt_Priority'Last);
```

In the above, both objects `DMA2_Stream0` and `DMA2_Stream5` are associated with the same object named `DMA2`, an instance of the `DMA_Controller` type. The difference in the objects is the stream that generates the interrupts they handle. One object handles `Stream_0` interrupts and the other handles those from `Stream_5`. Package `Ada.Interrupts.Names` for this target (for GNAT) declares distinct names for the streams and devices generating the interrupts, hence `DMA2_Stream0_Interrupt` and `DMA2_Stream5_Interrupt`.

On both objects the priority is the highest interrupt priority (and hence the highest overall), `Interrupt_Priority'Last`. That will work, but of course all interrupts will be blocked during the execution of the handler, as well as the execution of any other subprogram or entry in the same PO. That means that the clock interrupt is blocked for that interval, for example. We use that interrupt value in our demonstrations for expedience, but in a real application you'd almost certainly use a lower value specific to the interrupt handled.

We could reduce the number of discriminants, and also make the code more robust, by taking advantage of the requirement that type `Interrupt_Id` be a discrete type. As such, it can be used as the index type into arrays. Here is a driver example with only the `Interrupt_Id` discriminant required:

```
Device_Priority : constant array (Interrupt_Id) of Interrupt_Priority := ( ... );

protected type Device_Interface
  (IRQ : Interrupt_Id)
with
  Interrupt_Priority => Device_Priority (IRQ)
is
  procedure Handler with Attach_Handler => IRQ;
  ...
end Device_Interface;
```

Now we use the one IRQ discriminant both to assign the priorities for distinct objects and to attach their handler procedures.

79.7.2 Multi-Level Handlers

Interrupt handlers are intended to be very brief, in part because they prevent lower priority interrupts and application tasks from executing.

However, complete interrupt processing may require more than just the short protected procedure handler's activity. Therefore, two levels of handling are common: the protected procedure interrupt handler and a task. The handler does the least possible and then signals the task to do the rest.

Of course, sometimes the handler does everything required and just needs to signal the application. In that case, the awakened task does no further "interrupt processing" but simply uses the result.

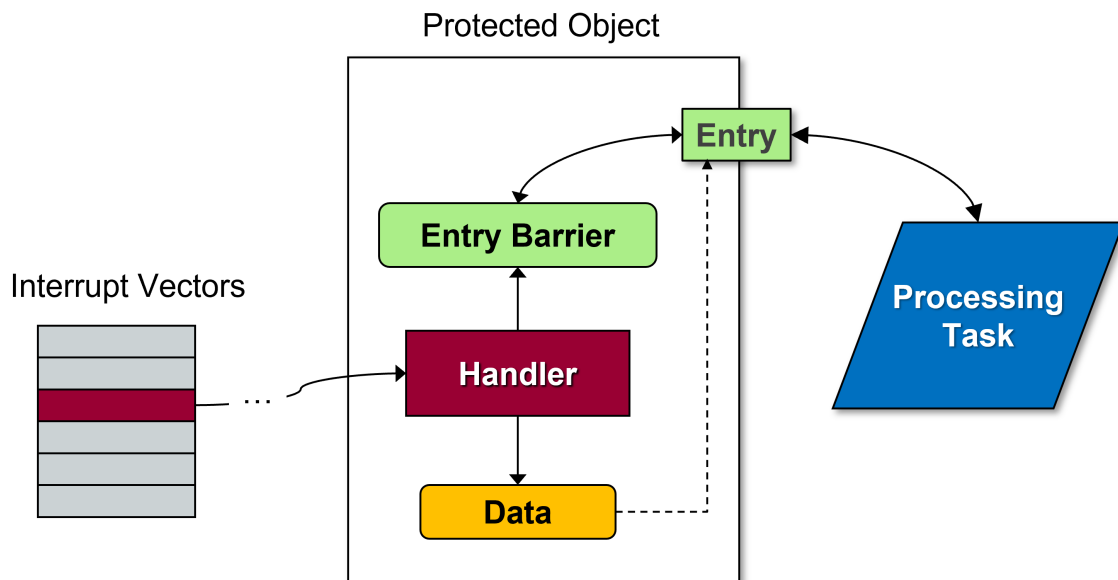
Regardless, the same issues apply: 1) How do application tasks synchronize with the handlers? Assuming the task is not polling the event, at some point the task must stop what

it was doing and suspend, waiting for the handler to signal it. 2) Once synchronized, how can the handlers pass data to the tasks?

Using protected objects for interrupt handling provides an efficient mechanism that elegantly addresses both issues. In addition, when data communication is not required, another standard language mechanism is available. These give rise to two design idioms. We will explore both.

In the first idiom, the protected object contains a protected entry as well as the interrupt handler procedure. The task suspends on the entry when ready for the handler results, controlled by the barrier condition as usual. The protected handler procedure responds to interrupts, managing data (if any) as required. When ready, based on what the handler does, the handler sets the entry barrier to **True**. That allows the suspended task to execute the entry body. The entry body can do whatever is required, possibly just copying the local protected data to the entry parameters. Of course, the entry may be used purely for synchronizing with the handler, i.e., suspending and resuming the task, in which case there would be no parameters passed.

The image below depicts this design.



The DMA_Interrupt_Controller described earlier actually uses this design.

```

protected type DMA_Interrupt_Controller
  (Controller : not null access DMA_Controller;
   Stream    : DMA_Stream_Selector;
   IRQ       : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System.Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is
  procedure Start_Transfer
    (Source      : Address;
     Destination : Address;
     Data_Count  : UInt16);

  procedure Abort_Transfer (Result : out DMA_Error_Code);
  
```

(continues on next page)

(continued from previous page)

```

procedure Clear_Transfer_State;

function Buffer_Error return Boolean;

entry Wait_For_Completion (Status : out DMA_Error_Code);

private

  procedure Interrupt_Handler with Attach_Handler => IRQ;

  No_Transfer_In_Progress : Boolean := True;
  Last_Status              : DMA_Error_Code := DMA_No_Error;
  Had_Buffer_Error         : Boolean := False;

end DMA_Interrupt_Controller;

```

The client application code (task) calls procedure `Start_Transfer` to initiate the DMA transaction, then presumably goes off to accomplish something else, and eventually calls the `Wait_For_Completion` entry. That call blocks the task if the device has not yet completed the DMA transfer. The interrupt handler procedure, cleverly named `Interrupt_Handler`, handles the interrupts, one of which indicates that the transfer has completed. Device errors also generate interrupts so the handler detects them and acts accordingly. Eventually, the handler sets the barrier to **True** and the task can get the status via the entry parameter.

```

procedure Start_Transfer
  (Source      : Address;
   Destination : Address;
   Data_Count  : UInt16)
is
begin
  No_Transfer_In_Progress := False;
  Had_Buffer_Error := False;
  Clear_All_Status (Controller.all, Stream);
  Start_Transfer_with_Interrupts
    (Controller.all,
     Stream,
     Source,
     ...,
     Enabled_Interrupts =>
       (Half_Transfer_Complete_Interrupt => False,
        others => True));
end Start_Transfer;

entry Wait_For_Completion
  (Status : out DMA_Error_Code)
when
  No_Transfer_In_Progress
is
begin
  Status := Last_Status;
end Wait_For_Completion;

```

In the above, the entry barrier consists of the Boolean variable `No_Transfer_In_Progress`. Procedure `Start_Transfer` first sets that variable to **False** so that a caller to `Wait_For_Completion` will suspend until the transaction completes one way or the other. Eventually, the handler sets `No_Transfer_In_Progress` to **True**.

```

procedure Interrupt_Handler is

```

(continues on next page)

(continued from previous page)

```

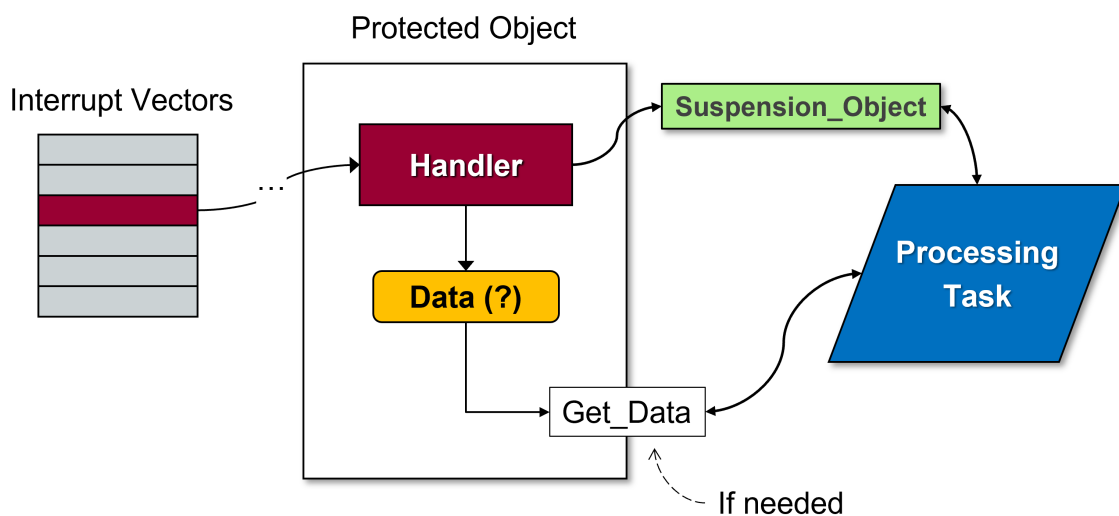
subtype Checked_Status_Flag is DMA_Status_Flag with
  Static_Predicate => Checked_Status_Flag /= Half_Transfer_Complete_Indicated;
begin
  for Flag in Checked_Status_Flag loop
    if Status (Controller.all, Stream, Flag) then
      case Flag is
        when FIFO_Error_Indicated =>
          Last_Status := DMA_FIFO_Error;
          Had_Buffer_Error := True;
          No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
        when Direct_Mode_Error_Indicated =>
          Last_Status := DMA_Direct_Mode_Error;
          No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
        when Transfer_Error_Indicated =>
          Last_Status := DMA_Transfer_Error;
          No_Transfer_In_Progress := True;
        when Transfer_Complete_Indicated =>
          Last_Status := DMA_No_Error;
          No_Transfer_In_Progress := True;
      end case;
      Clear_Status (Controller.all, Stream, Flag);
    end if;
  end loop;
end Interrupt_Handler;

```

This device driver doesn't bother with interrupts indicating that transfers are half-way complete so that specific status flag is ignored. In response to an interrupt, the handler checks each status flag to determine what happened. Note the resulting assignments for both the protected variables `Last_Status` and `No_Transfer_In_Progress`. The variable `No_Transfer_In_Progress` controls the entry, and `Last_Status` is passed to the caller via the entry formal parameter. When the interrupt handler exits, the resulting protected action allows the now-enabled entry call to execute.

In the second design idiom, the handler again synchronizes with the application task, but not using a protected entry.

The image below depicts this design.



In this approach, the task synchronizes with the handler using a `Suspension_Object` variable. The type `Suspension_Object` is defined in the language standard package `Ada.Synchronous_Task_Control`. Essentially, the type provides a thread-safe Boolean flag. Callers can suspend themselves (hence the package name) until another task resumes them by setting the flag to **True**. Here's the package declaration, somewhat elided:

```
package Ada.Synchronous_Task_Control is
    type Suspension_Object is limited private;
    procedure Set_True (S : in out Suspension_Object);
    procedure Set_False (S : in out Suspension_Object);
    function Current_State (S : Suspension_Object) return Boolean;
    procedure Suspend_Until_True (S : in out Suspension_Object);
private
    ...
end Ada.Synchronous_Task_Control;
```

Tasks call `Suspend_Until_True` to suspend themselves on some object of the type passed as the parameter. The call suspends the caller until that object becomes **True**. If it is already **True**, the caller continues immediately. Objects of type `Suspension_Object` are automatically set to **False** initially, and become **True** via a call to `Set_True`. As part of the return from a call to `Suspend_Until_True`, the flag is set back to **False**. As a result, you probably only need those two subprograms.

The interrupt handler procedure responds to interrupts, eventually setting some visible `Suspension_Object` to **True** so that the caller will be signaled and resume. Here's an example showing both the protected object, with handler, and a `Suspension_Object` declaration:

```
with Ada.Interrupts.Names;           use Ada.Interrupts.Names;
with Ada.Synchronous_Task_Control;   use Ada.Synchronous_Task_Control;

package Gyro_Interrupts is
    Data_Available : Suspension_Object;

    protected Handler is
        pragma Interrupt_Priority;
    private
        procedure IRQ_Handler
            with Attach_Handler => EXTI2_Interrupt;
        end Handler;
end Gyro_Interrupts;
```

In the code above, `Gyro_Interrupts.Data_Available` is the `Suspension_Object` variable visible both to the interrupt handler PO and the client task.

`EXTI2_Interrupt` is "external interrupt number 2" on this particular microcontroller. It is connected to an external device, not on the SoC itself. Specifically, it is connected to a [L3GD20 MEMS motion sensor](https://www.st.com/en/mems-and-sensors/l3gd20.html)⁴³⁰, a three-axis digital output gyroscope. This gyroscope can be either polled or generate interrupts when ever data are available. The handler is very simple:

⁴³⁰ <https://www.st.com/en/mems-and-sensors/l3gd20.html>

```

with STM32.EXTI; use STM32.EXTI;

package body Gyro_Interrupts is
    protected body Handler is

        procedure IRQ_Handler is
        begin
            if External_Interrupt_Pending (EXTI_Line_2) then
                Clear_External_Interrupt (EXTI_Line_2);
                Set_True (Data_Available);
            end if;
        end IRQ_Handler;

    end Handler;

end Gyro_Interrupts;

```

The handler simply clears the interrupt and resumes the caller task via a call to `Set_True` on the variable declared in the package spec.

The lack of an entry means that no data can be passed to the task via entry parameters. It is possible to pass data to the task but doing so would require an additional protected procedure or function.

The gyroscope hardware device interface is in package `L3GD20`. Here are the pertinent parts:

```

package L3GD20 is

    type Three_Axis_Gyroscope is tagged limited private;

    procedure Initialize
        (This      : in out Three_Axis_Gyroscope;
         Port      : Any_SPI_Port;
         Chip_Select : Any_GPIO_Point);

    ...

    procedure Enable_Data_Ready_Interrupt (This : in out Three_Axis_Gyroscope);

    ...

    type Angle_Rate is new Integer_16;

    type Angle_Rates is record
        X : Angle_Rate; -- pitch, per Figure 2, pg 7 of the Datasheet
        Y : Angle_Rate; -- roll
        Z : Angle_Rate; -- yaw
    end record with Size => 3 * 16;

    ...

    procedure Get_Raw_Angle_Rates
        (This : Three_Axis_Gyroscope;
         Rates : out Angle_Rates);

    ...

end L3GD20;

```

With those packages available, we can write a simple main program to use the gyro. The real demo displayed the readings on an LCD but we've elided all those irrelevant details:

```
with Gyro_Interrupts;
with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
with L3GD20; use L3GD20;
with STM32.Board;
...

procedure Demo_L3GD20 is

  Axes : L3GD20.Angle_Rates;

  ...

  procedure Await_Raw_Angle_Rates (Rates : out L3GD20.Angle_Rates) is
  begin
    Suspend_Until_True (Gyro_Interrupts.Data_Available);
    L3GD20.Get_Raw_Angle_Rates (STM32.Board.Gyro, Rates);
  end Await_Raw_Angle_Rates;

  ...

begin
  Configure_Gyro;
  Configure_Gyro_Interrupt;
  ...
  loop
    Await_Raw_Angle_Rates (Axes);
    ...
  end loop;
end Demo_L3GD20;
```

The demo is a main procedure, even though we've been describing the client application code in terms of tasks. The main procedure is executed by the implicit "environment task" so it all still works. `Await_Raw_Angle_Rates` suspends (if necessary) on `Gyro_Interrupts.Data_Available` and then calls `L3GD20.Get_Raw_Angle_Rates` to get the rate values.

The operations provided by `Suspension_Object` are faster than protected entries, and noticeably so. However, that performance difference is due to the fact that `Suspension_Object` provides so much less capability than entries. In particular, there is no notion of protected actions, nor expressive entry barriers for condition synchronization, nor parameters to pass data while synchronized. Most importantly, there is no caller queue, so at most one caller can be waiting at a time on any given `Suspension_Object` variable. You'll get `Program_Error` if you try. Protected entries should be your first design choice. Note that the Ravenscar restrictions can make use of `Suspension_Object` much more likely.

79.8 Final Points

As you can see, the semantics of protected objects are a good fit for interrupt handling. However, other forms of handlers are allowed to be supported. For example, the compiler and RTL for a specific target may include support for interrupts generated by a device known to be available with that target. For illustration, let's imagine the target always has a serial port backed by a UART. In addition to handlers as protected procedure without parameters, perhaps the compiler and RTL support interrupt handlers with a single parameter of type `Unsigned_8` (or larger) as supported by the UART.

Overall, the interrupt model defined and supported by Ada is quite close to the canonical model presented by most programming languages, in part because it matches the model presented by typical hardware.

CONCLUSION

In the introduction to this course, we defined an "embedded system" as a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system. That, in itself, sets this kind of programming apart from the more typical host-oriented programming. But the context also implies fewer resources are available, especially memory and electrical power, as well as processor power. Add to those limitations a frequent reliability requirement and you have a demanding context for development.

Using Ada can help you in this context, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal in this course was to get you started in using it well.

To that end, we spent a lot of time talking about how to use Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data. Ada has a lot of support for this activity so there was much to explore.

Likewise, we examined development using Ada in combination with other languages, a not uncommon approach. Specifically, we saw how to interface with code and data written in other languages, and how (and why) to work with assembly language. Development in just one language is becoming less common over time so these were important aspects to know.

One of the more distinctive activities of embedded programming involves interacting with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. (This can be one of the more entertaining activities as well.) We covered how to interact with these memory-mapped devices using representation specifications, data structures that simplified the functional code, and time-honored aspects of software engineering, including abstract data types.

Finally, we explored how to handle interrupts in Ada, another distinctive part of embedded systems programming. As we saw, Ada has extensive support for handling interrupts, using the same building blocks — protected objects — used in concurrent programming. These constructs provide a way to handle interrupts that is as portable as possible, in what is otherwise a very hardware-specific endeavor.

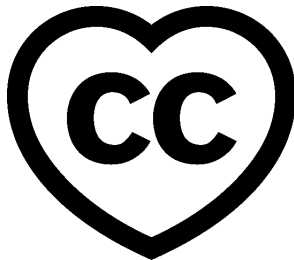
In the course, we mentioned a library of freely-available device drivers in Ada known as the Ada Driver Library (ADL). The ADL is a good resource for learning how Ada can be used to develop software for embedded systems using real-world devices and processors. Becoming familiar with it would be a good place to go next. Contributing to it would be even better! The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

Part VIII

Ada for the Embedded C Developer

Copyright © 2020 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)⁴³¹



This course introduces you to the Ada language by comparing it to C. It assumes that you have good knowledge of the C language. It also assumes that the choice of learning Ada is guided by considerations linked to reliability, safety or security. In that sense, it teaches you Ada paradigms that should be applied in replacement of those usually applied in C.

This course also introduces you to the SPARK subset of the Ada programming language, which removes a few features of the language with undefined behavior, so that the code is fit for sound static analysis techniques.

This course was written by Quentin Ochem, Robert Tice, Gustavo A. Hoffmann, and Patrick Rogers and reviewed by Patrick Rogers, Filip Gajowniczek, and Tucker Taft.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](#)⁴³². The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;

⁴³¹ <http://creativecommons.org/licenses/by-sa/4.0>

3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

⁴³² https://learn.adacore.com/zip/learning-ada_code.zip

INTRODUCTION

81.1 So, what is this Ada thing anyway?

To answer this question let's introduce Ada as it compares to C for an embedded application. C developers are used to a certain coding semantic and style of programming. Especially in the embedded domain, developers are used to working at a very low level near the hardware to directly manipulate memory and registers. Normal operations involve mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations as it is a low level language that was designed to replace assembly language for faster, more efficient programming. Because of this minimal abstraction, the programmer has to model the data that represents the problem they are trying to solve using the language of the physical hardware.

Let's look at an example of this problem in action by comparing the same program in Ada and C:

[C]

Listing 2077: main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define DEGREES_MAX      (360)
5  typedef unsigned int degrees;
6
7  #define MOD_DEGREES(x)   (x % DEGREES_MAX)
8
9  degrees add_angles(degrees* list, int length)
10 {
11     degrees sum = 0;
12     for(int i = 0; i < length; ++i) {
13         sum += list[i];
14     }
15
16     return sum;
17 }
18
19 int main(int argc, char** argv)
20 {
21     degrees list[argc - 1];
22
23     for(int i = 1; i < argc; ++i) {
24         list[i - 1] = MOD_DEGREES(atoi(argv[i]));
25     }
26
27     printf("Sum: %d\n", add_angles(list, argc - 1));
28
```

(continues on next page)

(continued from previous page)

```
29  return 0;  
30 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Introduction.Add_Angles_C
MD5: a6d184caaec372c538634c578b5e144b

Runtime output

Sum: 0

[Ada]

Listing 2078: sum_angles.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;  
2  with Ada.Text_IO; use Ada.Text_IO;  
3  
4  procedure Sum_Angles is  
5  
6      DEGREES_MAX : constant := 360;  
7      type Degrees is mod DEGREES_MAX;  
8  
9      type Degrees_List is array (Natural range <>) of Degrees;  
10  
11     function Add_Angles (List : Degrees_List) return Degrees  
12     is  
13         Sum : Degrees := 0;  
14         begin  
15             for I in List'Range loop  
16                 Sum := Sum + List (I);  
17             end loop;  
18  
19             return Sum;  
20         end Add_Angles;  
21  
22     List : Degrees_List (1 .. Argument_Count);  
23     begin  
24         for I in List'Range loop  
25             List (I) := Degrees (Integer'Value (Argument (I)));  
26         end loop;  
27  
28         Put_Line ("Sum:" & Add_Angles (List)'Img);  
29     end Sum_Angles;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Introduction.Add_Angles_Ada
MD5: b5a446e5c27aa18c917ae8c2cc6c1605

Runtime output

Sum: 0

Here we have a piece of code in C and in Ada that takes some numbers from the command line and stores them in an array. We then sum all of the values in the array and print the result. The tricky part here is that we are working with values that model an angle in degrees. We know that angles are modular types, meaning that angles greater than 360° can also be represented as `Angle mod 360`. So if we have an angle of 400° , this is equivalent to 40° . In order to model this behavior in C we had to create the `MOD_DEGREES`

macro, which performs the modulus operation. As we read values from the command line, we convert them to integers and perform the modulus before storing them into the array. We then call `add_angles` which returns the sum of the values in the array. Can you spot the problem with the C code?

Try running the Ada and C examples using the input sequence `340 2 50 70`. What does the C program output? What does the Ada program output? Why are they different?

The problem with the C code is that we forgot to call `MOD_DEGREES` in the for loop of `add_angles`. This means that it is possible for `add_angles` to return values greater than `DEGREES_MAX`. Let's look at the equivalent Ada code now to see how Ada handles the situation. The first thing we do in the Ada code is to create the type `Degrees` which is a modular type. This means that the compiler is going to handle performing the modulus operation for us. If we use the same for loop in the `Add_Angles` function, we can see that we aren't doing anything special to make sure that our resulting value is within the 360° range we need it to be in.

The takeaway from this example is that Ada tries to abstract some concepts from the developer so that the developer can focus on solving the problem at hand using a data model that models the real world rather than using data types prescribed by the hardware. The main benefit of this is that the compiler takes some responsibility from the developer for generating correct code. In this example we forgot to put in a check in the C code. The compiler inserted the check for us in the Ada code because we told the compiler what we were trying to accomplish by defining strong types.

Ideally, we want all the power that the C programming language can give us to manipulate the hardware we are working on while also allowing us the ability to more accurately model data in a safe way. So, we have a dilemma; what can give us the power of operations like the C language, but also provide us with features that can minimize the potential for developer error? Since this course is about Ada, it's a good bet we're about to introduce the Ada language as the answer to this question...

Unlike C, the Ada language was designed as a higher level language from its conception; giving more responsibility to the compiler to generate correct code. As mentioned above, with C, developers are constantly shifting, masking, and accessing bits directly on memory pointers. In Ada, all of these operations are possible, but in most cases, there is a better way to perform these operations using higher level constructs that are less prone to mistakes, like off-by-one or unintentional buffer overflows. If we were to compare the same application written using C and with Ada using high level constructs, we would see similar performance in terms of speed and memory efficiency. If we compare the object code generated by both compilers, it's possible that they even look identical!

81.2 Ada — The Technical Details

Like C, Ada is a compiled language. This means that the compiler will parse the source code and emit machine code native to the target hardware. The Ada compiler we will be discussing in this course is the GNAT compiler. This compiler is based on the GCC technology like many C and C++ compilers available. When the GNAT compiler is invoked on Ada code, the GNAT front-end expands and translates the Ada code into an intermediate language which is passed to GCC where the code is optimized and translated to machine code. A C compiler based on GCC performs the same steps and uses the same intermediate GCC representation. This means that the optimizations we are used to seeing with a GCC based C compiler can also be applied to Ada code. The main difference between the two compilers is that the Ada compiler is expanding high level constructs into intermediate code. After expansion, the Ada code will be very similar to the equivalent C code.

It is possible to do a line-by-line translation of C code to Ada. This feels like a natural step for a developer used to C paradigms. However, there may be very little benefit to doing so. For the purpose of this course, we're going to assume that the choice of Ada over C is guided by

considerations linked to reliability, safety or security. In order to improve upon the reliability, safety and security of our application, Ada paradigms should be applied in replacement of those usually applied in C. Constructs such as pointers, preprocessor macros, bitwise operations and defensive code typically get expressed in Ada in very different ways, improving the overall reliability and readability of the applications. Learning these new ways of coding, often, requires effort by the developer at first, but proves more efficient once the paradigms are understood.

In this course we will also introduce the SPARK subset of the Ada programming language. The SPARK subset removes a few features of the language, i.e., those that make proof difficult, such as pointer aliasing. By removing these features we can write code that is fit for sound static analysis techniques. This means that we can run mathematical provers on the SPARK code to prove certain safety or security properties about the code.

THE C DEVELOPER'S PERSPECTIVE ON ADA

82.1 What we mean by Embedded Software

The Ada programming language is a general programming language, which means it can be used for many different types of applications. One type of application where it particularly shines is reliable and safety-critical embedded software; meaning, a platform with a microprocessor such as ARM, PowerPC, x86, or RISC-V. The application may be running on top of an embedded operating system, such as an embedded Linux, or directly on bare metal. And the application domain can range from small entities such as firmware or device controllers to flight management systems, communication based train control systems, or advanced driver assistance systems.

82.2 The GNAT Toolchain

The toolchain used throughout this course is called GNAT, which is a suite of tools with a compiler based on the GCC environment. It can be obtained from AdaCore, either as part of a commercial contract with [GNAT Pro](#)⁴³³ or at no charge with the [GNAT Community edition](#)⁴³⁴. The information in this course will be relevant no matter which edition you're using. Most examples will be runnable on the native Linux or Windows version for convenience. Some will only be relevant in the context of a cross toolchain, in which case we'll be using the embedded ARM bare metal toolchain.

As for any Ada compiler, GNAT takes advantage of implementation permissions and offers a project management system. Because we're talking about embedded platforms, there are a lot of topics that we'll go over which will be specific to GNAT, and sometimes to specific platforms supported by GNAT. We'll try to make the distinction between what is GNAT-specific and Ada generic as much as possible throughout this course.

For an introduction to the GNAT Toolchain for the GNAT Community edition, you may refer to the [Introduction to GNAT Toolchain](#) (page 2127) course.

82.3 The GNAT Toolchain for Embedded Targets

When we're discussing embedded programming, our target device is often different from the host, which is the device we're using to actually write and build an application. In this case, we're talking about cross compilation platforms (concisely referred to as cross platforms).

The GNAT toolchain supports cross platform compilation for various target devices. This section provides a short introduction to the topic. For more details, please refer to the [GNAT User's Guide Supplement for Cross Platforms](#)⁴³⁵

⁴³³ <https://www.adacore.com/gnatpro>

⁴³⁴ <https://www.adacore.com/community>

⁴³⁵ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html

GNAT supports two types of cross platforms:

- **cross targets**, where the target device has an embedded operating system.
 - ARM-Linux, which is commonly found in a Raspberry-Pi, is a prominent example.
- **bareboard targets**, where the run-times do not depend on an operating system.
 - In this case, the application has direct access to the system hardware.

For each platform, a set of run-time libraries is available. Run-time libraries implement a subset of the Ada language for different use cases, and they're different for each target platform. They may be selected via an attribute in the project's GPR project file or as a command-line switch to **GPRbuild**. Although the run-time libraries may vary from target to target, the user interface stays the same, providing portability for the application.

Run-time libraries consists of:

1. Files that are dependent on the target board.
 - These files are responsible for configuring and interacting with the hardware.
 - They are known as a Board Support Package — commonly referred to by their abbreviation *BSP*.
2. Code that is target-independent.
 - This code implements language-defined functionality.

The bareboard run-time libraries are provided as customized run-times that are configured to target a very specific micro-controller or processor. Therefore, for different micro-controllers and processors, the run-time libraries need to be ported to the specific target. These are some examples of what needs to be ported:

- startup code / scripts;
- clock frequency initializations;
- memory mapping / allocation;
- interrupts and interrupt priorities;
- register descriptions.

For more details on the topic, please refer to the following chapters of the [GNAT User's Guide Supplement for Cross Platforms](#)⁴³⁶:

- [Bareboard Topics](#)⁴³⁷
- [Customized Run-Time Libraries](#)⁴³⁸

82.4 Hello World in Ada

The first piece of code to translate from C to Ada is the usual Hello World program:

[C]

Listing 2079: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     printf("Hello World\n");
```

(continues on next page)

⁴³⁶ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html

⁴³⁷ http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/bareboard_topics.html

⁴³⁸ http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/customized_run-time_libraries.html

(continued from previous page)

```

6   return 0;
7 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Hello_World_C
MD5: 59685c72296a032893cda71dade24196

Runtime output

Hello World

[Ada]

Listing 2080: hello_world.adb

```

1 with Ada.Text_IO;
2
3 procedure Hello_World
4 is
5 begin
6     Ada.Text_IO.Put_Line ("Hello World");
7 end Hello_World;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Hello_World_Ada
MD5: f1a7c6a4fd679c4caea7ee31d14aab2e

Runtime output

Hello World

The resulting program will print Hello World on the screen. Let's now dissect the Ada version to describe what is going on:

The first line of the Ada code is giving us access to the `Ada.Text_IO` library which contains the `Put_Line` function we will use to print the text to the console. This is similar to C's `#include <stdio.h>`. We then create a procedure which executes `Put_Line` which prints to the console. This is similar to C's `printf` function. For now, we can assume these Ada and C features have similar functionality. In reality, they are very different. We will explore that more as we delve further into the Ada language.

You may have noticed that the Ada syntax is more verbose than C. Instead of using braces `{}` to declare scope, Ada uses keywords. `is` opens a declarative scope — which is empty here as there's no variable to declare. `begin` opens a sequence of statements. Within this sequence, we're calling the function `Put_Line`, prefixing explicitly with the name of the library unit where it's declared, `Ada.Text_IO`. The absence of the end of line `\n` can also be noted, as `Put_Line` always terminates by an end of line.

82.5 The Ada Syntax

Ada syntax might seem peculiar at first glance. Unlike many other languages, it's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a more explicit language — its syntax was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner. For example:

- full words like `begin` and `end` are used in place of curly braces.

- Conditions are written using **if**, **then**, **elsif**, **else**, and **end if**.
- Ada's assignment operator does not double as an expression, eliminating potential mistakes that could be caused by `=` being used where `==` should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C. Multi line comments like C's `/* */` do not exist in Ada.

Ada compilers are stricter with type and range checking than most C programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, checks (such as array bounds checks) provide verification that could not be done at compile time but can be performed either at run-time, or through formal proof (with the SPARK tooling).

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same identifier; likewise **begin**, **BEGIN**, **Begin**, etc. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

82.6 Compilation Unit Structure

Both C and Ada were designed with the idea that the code specification and code implementation could be separated into two files. In C, the specification typically lives in the `.h`, or header file, and the implementation lives in the `.c` file. Ada is superficially similar to C. With the GNAT toolchain, compilation units are stored in files with an `.ads` extension for specifications and with an `.adb` extension for implementations.

One main difference between the C and Ada compilation structure is that Ada compilation units are structured into something called packages.

82.7 Packages

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C. A specification defines a package and the implementation implements the package. We saw this in an earlier example when we included the `Ada.Text_IO` package into our application. The package specification has the structure:

[Ada]

```
-- my_package.ads
package My_Package is

    -- public declarations

private

    -- private declarations

end My_Package;
```

The package implementation, or body, has the structure:

```
-- my_package.adb
package body My_Package is

    -- implementation

end My_Package;
```

82.7.1 Declaration Protection

An Ada package contains three parts that, for GNAT, are separated into two files: `.ads` files contain public and private Ada specifications, and `.adb` files contain the implementation, or Ada bodies.

[Ada]

```
package Package_Name is
    -- public specifications
private
    -- private specifications
end Package_Name;

package body Package_Name is
    -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. Another use-case is the prevention of package users from accessing package state/data arbitrarily. The private reserved word splits the package spec into *public* and *private* parts. For example:

[Ada]

Listing 2081: types.ads

```
1 package Types is
2     type Type_1 is private;
3     type Type_2 is private;
4     type Type_3 is private;
```

(continues on next page)

(continued from previous page)

```
5   procedure P (X : Type_1);
6   -- ...
7 private
8   procedure Q (Y : Type_1);
9   type Type_1 is new Integer range 1 .. 1000;
10  type Type_2 is array (Integer range 1 .. 1000) of Integer;
11  type Type_3 is record
12    A, B : Integer;
13  end record;
14 end Types;
```

Code block metadata

```
Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Private_Types
MD5: ae4a9e4d10b55e7efd92d7952ba22f4f
```

Subprograms declared above the **private** separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts. A package specification does not require a private section.

82.7.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

[Ada]

```
-- root-child.ads

package Root.Child is
  -- package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
  -- package body goes here
end Root.Child;
```

Here, `Root.Child` is a child package of `Root`. The public part of `Root.Child` has access to the public part of `Root`. The private part of `Child` has access to the private part of `Root`, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

82.7.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a **with** clause that references the package, which is similar to the C `#include` directive. After a **with** clause makes a package available, references to the package contents require the name of the package as a prefix, with a dot after the package name. This prefix can be omitted if a **use** clause is employed.

[Ada]

Listing 2082: pck.ads

```

1  -- pck.ads
2
3  package Pck is
4      My_Glob : Integer;
5  end Pck;
```

Listing 2083: main.adb

```

1  -- main.adb
2
3  with Pck;
4
5  procedure Main is
6  begin
7      Pck.My_Glob := 0;
8  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Using_Pkg_Entities
MD5: 4215ba710eb54478538dc001bb74ce09

In contrast to C, the Ada **with** clause is a *semantic inclusion* mechanism rather than a *text inclusion* mechanism; for more information on this difference please refer to [Packages](#) (page 35).

82.8 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[C]

Listing 2084: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      // variable declarations
6      int a = 0, b = 0, c = 100, d;
7
8      // c shorthand for increment
9      a++;
10
11     // regular addition
12     d = a + b + c;
13
14     // printing the result
15     printf("d = %d\n", d);
16
17     return 0;
18 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_C
MD5: ba258dac5c052a97da475239e2f2ce96

Runtime output

```
d = 101
```

[Ada]

Listing 2085: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main
4 is
5     -- variable declaration
6     A, B : Integer := 0;
7     C    : Integer := 100;
8     D    : Integer;
9 begin
10    -- Ada does not have a shortcut format for increment like in C
11    A := A + 1;
12
13    -- regular addition
14    D := A + B + C;
15
16    -- printing the result
17    Ada.Text_IO.Put_Line ("D =" & D'Img);
18 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_Ada
MD5: eaff76f36d5f938bd806d29048df7865

Runtime output

```
D = 101
```

You'll notice that, in both languages, statements are terminated with a semicolon. This means that you can have multi-line statements.

The shortcuts of incrementing and decrementing

You may have noticed that Ada does not have something similar to the `a++` or `a--` operators. Instead you must use the full assignment `A := A + 1` or `A := A - 1`.

In the Ada example above, there are two distinct sections to the `procedure Main`. This first section is delimited by the `is` keyword and the `begin` keyword. This section is called the declarative block of the subprogram. The declarative block is where you will define all the local variables which will be used in the subprogram. C89 had something similar, where developers were required to declare their variables at the top of the scope block. Most C developers may have run into this before when trying to write a for loop:

[C]

Listing 2086: main.c

```
1 /* The C89 version */
2
3 #include <stdio.h>
4
5 int average(int* list, int length)
```

(continues on next page)

(continued from previous page)

```

6 {
7     int i;
8     int sum = 0;
9
10    for(i = 0; i < length; ++i) {
11        sum += list[i];
12    }
13    return (sum / length);
14 }
15
16 int main(int argc, const char * argv[])
17 {
18     int vals[] = { 2, 2, 4, 4 };
19
20     printf("Average: %d\n", average(vals, 4));
21
22     return 0;
23 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Average_C89
MD5: 5c89aa28cba0bae4d963b235c53aedef2

Runtime output

Average: 3

[C]

Listing 2087: main.c

```

1  // The modern C way
2
3  #include <stdio.h>
4
5  int average(int* list, int length)
6  {
7      int sum = 0;
8
9      for(int i = 0; i < length; ++i) {
10         sum += list[i];
11     }
12
13     return (sum / length);
14 }
15
16 int main(int argc, const char * argv[])
17 {
18     int vals[] = { 2, 2, 4, 4 };
19
20     printf("Average: %d\n", average(vals, 4));
21
22     return 0;
23 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Average_C_Modern
MD5: 6354863137d78adb974743915d1d4530

Runtime output

```
Average: 3
```

For the fun of it, let's also see the Ada way to do this:

[Ada]

Listing 2088: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4   type Int_Array is array (Natural range <>) of Integer;
5
6   function Average (List : Int_Array) return Integer
7   is
8     Sum : Integer := 0;
9   begin
10    for I in List'Range loop
11      Sum := Sum + List (I);
12    end loop;
13
14    return (Sum / List'Length);
15  end Average;
16
17  Vals : constant Int_Array (1 .. 4) := (2, 2, 4, 4);
18 begin
19   Ada.Text_IO.Put_Line ("Average: " & Integer'Image (Average (Vals)));
20 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Average_Ada
MD5: 52abb574d7a8b3bdb56715735dcd1d54

Runtime output

```
Average: 3
```

We will explore more about the syntax of loops in Ada in a future section of this course; but for now, notice that the `I` variable used as the loop index is not declared in the declarative section!

Declaration Flippy Floppy

Something peculiar that you may have noticed about declarations in Ada is that they are backwards from the way C does declarations. The C language expects the type followed by the variable name. Ada expects the variable name followed by a colon and then the type.

The next block in the Ada example is between the **begin** and **end** keywords. This is where your statements will live. You can create new scopes by using the **declare** keyword:

[Ada]

Listing 2089: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main
4 is
```

(continues on next page)

(continued from previous page)

```

5  -- variable declaration
6  A, B : Integer := 0;
7  C    : Integer := 100;
8  D    : Integer;
9  begin
10 -- Ada does not have a shortcut format for increment like in C
11 A := A + 1;
12
13 -- regular addition
14 D := A + B + C;
15
16 -- printing the result
17 Ada.Text_IO.Put_Line ("D =" & D'Img);
18
19 declare
20   E : constant Integer := D * 100;
21 begin
22   -- printing the result
23   Ada.Text_IO.Put_Line ("E =" & E'Img);
24 end;
25
26 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_Block_Ada
MD5: 9239b993a7eadb13a27bd3618a03431f

Runtime output

```
D = 101
E = 10100
```

Notice that we declared a new variable E whose scope only exists in our newly defined block. The equivalent C code is:

[C]

Listing 2090: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      // variable declarations
6      int a = 0, b = 0, c = 100, d;
7
8      // c shorthand for increment
9      a++;
10
11     // regular addition
12     d = a + b + c;
13
14     // printing the result
15     printf("d = %d\n", d);
16
17     {
18         const int e = d * 100;
19         printf("e = %d\n", e);
20     }
21
```

(continues on next page)

(continued from previous page)

```
22     return 0;  
23 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Var_Decl_Block_C
MD5: 1a837795575ddc026738d92c8655ab6c

Runtime output

```
d = 101  
e = 10100
```

Fun Fact about the C language assignment operator `=`: Did you know that an assignment in C can be used in an expression? Let's look at an example:

[C]

Listing 2091: main.c

```
1  #include <stdio.h>  
2  
3  int main(int argc, const char * argv[])  
4  {  
5      int a = 0;  
6  
7      if (a = 10)  
8          printf("True\n");  
9      else  
10         printf("False\n");  
11  
12     return 0;  
13 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Equal_C
MD5: 2d00ddf7e154cb888082c86b8fd36c58

Runtime output

```
True
```

Run the above code example. What does it output? Is that what you were expecting?

The author of the above code example probably meant to test if `a == 10` in the if statement but accidentally typed `=` instead of `==`. Because C treats assignment as an expression, it was able to evaluate `a = 10`.

Let's look at the equivalent Ada code:

[Ada]

Listing 2092: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;  
2  
3  procedure Main  
4  is  
5      A : Integer := 0;  
6  begin  
7
```

(continues on next page)

(continued from previous page)

```

8   if A := 10 then
9       Put_Line ("True");
10  else
11      Put_Line ("False");
12  end if;
13 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Equal_Ada
MD5: 1500b264531dfcc7a62eed2f22f511b

The above code will not compile. This is because Ada does not allow assignment as an expression.

i The "use" clause

You'll notice in the above code example, after **with** Ada.Text_IO; there is a new statement we haven't seen before — **use** Ada.Text_IO;. You may also notice that we are not using the Ada.Text_IO prefix before the Put_Line statements. When we add the use clause it tells the compiler that we won't be using the prefix in the call to subprograms of that package. The use clause is something to use with caution. For example: if we use the Ada.Text_IO package and we also have a Put_Line subprogram in our current compilation unit with the same signature, we have a (potential) collision!

82.9 Conditions

The syntax of an if statement:

[C]

Listing 2093: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      // try changing the initial value to change the
6      //   output of the program
7      int v = 0;
8
9      if (v > 0) {
10         printf("Positive\n");
11     }
12     else if (v < 0) {
13         printf("Negative\n");
14     }
15     else {
16         printf("Zero\n");
17     }
18
19     return 0;
20 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Condition_C
MD5: 69203e679085e73394d3620a5954262a

Runtime output

Zero

[Ada]

Listing 2094: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main
4  is
5      -- try changing the initial value to change the
6      -- output of the program
7      V : constant Integer := 0;
8  begin
9      if V > 0 then
10         Put_Line ("Positive");
11     elsif V < 0 then
12         Put_Line ("Negative");
13     else
14         Put_Line ("Zero");
15     end if;
16 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Condition_Ada
MD5: 417e557708472f9022db7d8c1ed6aa33

Runtime output

Zero

In Ada, everything that appears between the **if** and **then** keywords is the conditional expression, no parentheses are required. Comparison operators are the same except for:

Operator	C	Ada
Equality	==	=
Inequality	!=	/=
Not	!	not
And	&&	and
Or		or

The syntax of a switch/case statement:

[C]

Listing 2095: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      // try changing the initial value to change the
6      // output of the program
7      int v = 0;
8
9      switch(v) {
10         case 0:

```

(continues on next page)

(continued from previous page)

```

11     printf("Zero\n");
12     break;
13     case 1: case 2: case 3: case 4: case 5:
14     case 6: case 7: case 8: case 9:
15         printf("Positive\n");
16         break;
17     case 10: case 12: case 14: case 16: case 18:
18         printf("Even number between 10 and 18\n");
19         break;
20     default:
21         printf("Something else\n");
22         break;
23 }
24
25 return 0;
26 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_C
MD5: 1bdb3d0c151d71280ef9039841f7ee58

Runtime output

Zero

[Ada]

Listing 2096: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main
4  is
5      -- try changing the initial value to change the
6      -- output of the program
7      V : constant Integer := 0;
8  begin
9      case V is
10         when 0 =>
11             Put_Line ("Zero");
12         when 1 .. 9 =>
13             Put_Line ("Positive");
14         when 10 | 12 | 14 | 16 | 18 =>
15             Put_Line ("Even number between 10 and 18");
16         when others =>
17             Put_Line ("Something else");
18     end case;
19 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_Ada
MD5: 09e2318b56069281c95f23310dc121d1

Runtime output

Zero

Switch or Case?

A switch statement in C is the same as a case statement in Ada. This may be a little strange because C uses both keywords in the statement syntax. Let's make an analogy between C and Ada: C's **switch** is to Ada's **case** as C's **case** is to Ada's **when**.

Notice that in Ada, the case statement does not use the **break** keyword. In C, we use **break** to stop the execution of a case branch from falling through to the next branch. Here is an example:

[C]

Listing 2097: main.c

```
1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int v = 0;
6
7      switch(v) {
8          case 0:
9              printf("Zero\n");
10             case 1:
11                 printf("One\n");
12             default:
13                 printf("Other\n");
14         }
15
16         return 0;
17     }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Switch_Case_Break_C
MD5: fd0389205476f161655caf32244d9054

Runtime output

```
Zero
One
Other
```

Run the above code with `v = 0`. What prints? What prints when we change the assignment to `v = 1`?

When `v = 0` the program outputs the strings Zero then One then Other. This is called fall through. If you add the **break** statements back into the **switch** you can stop this fall through behavior from happening. The reason why fall through is allowed in C is to allow the behavior from the previous example where we want a specific branch to execute for multiple inputs. Ada solves this a different way because it is possible, or even probable, that the developer might forget a **break** statement accidentally. So Ada does not allow fall through. Instead, you can use Ada's syntax to identify when a specific branch can be executed by more than one input. If you want a range of values for a specific branch you can use the `First .. Last` notation. If you want a few non-consecutive values you can use the `Value1 | Value2 | Value3` notation.

Instead of using the word **default** to denote the catch-all case, Ada uses the **others** keyword.

82.10 Loops

Let's start with some syntax:

[C]

Listing 2098: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int v;
6
7      // this is a while loop
8      v = 1;
9      while(v < 100) {
10         v *= 2;
11     }
12     printf("v = %d\n", v);
13
14     // this is a do while loop
15     v = 1;
16     do {
17         v *= 2;
18     } while(v < 200);
19     printf("v = %d\n", v);
20
21     // this is a for loop
22     v = 0;
23     for(int i = 0; i < 5; ++i) {
24         v += (i * i);
25     }
26     printf("v = %d\n", v);
27
28     // this is a forever loop with a conditional exit
29     v = 0;
30     while(1) {
31         // do stuff here
32         v += 1;
33         if(v == 10)
34             break;
35     }
36     printf("v = %d\n", v);
37
38     // this is a loop over an array
39     {
40         #define ARR_SIZE (10)
41         const int arr[ARR_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
42         int sum = 0;
43
44         for(int i = 0; i < ARR_SIZE; ++i) {
45             sum += arr[i];
46         }
47         printf("sum = %d\n", sum);
48     }
49
50     return 0;
51 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loops_C
MD5: bcd8963884e2b2a5e364219f9b6b8fbc

Runtime output

```
v = 128
v = 256
v = 30
v = 10
sum = 55
```

[Ada]

Listing 2099: main.adb

```
1  with Ada.Text_IO;
2
3  procedure Main is
4      V : Integer;
5  begin
6      -- this is a while loop
7      V := 1;
8      while V < 100 loop
9          V := V * 2;
10     end loop;
11     Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));
12
13     -- Ada doesn't have an explicit do while loop
14     -- instead you can use the loop and exit keywords
15     V := 1;
16     loop
17         V := V * 2;
18         exit when V >= 200;
19     end loop;
20     Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));
21
22     -- this is a for loop
23     V := 0;
24     for I in 0 .. 4 loop
25         V := V + (I * I);
26     end loop;
27     Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));
28
29     -- this is a forever loop with a conditional exit
30     V := 0;
31     loop
32         -- do stuff here
33         V := V + 1;
34         exit when V = 10;
35     end loop;
36     Ada.Text_IO.Put_Line ("V = " & Integer'Image (V));
37
38     -- this is a loop over an array
39     declare
40         type Int_Array is array (Natural range 1 .. 10) of Integer;
41
42         Arr : constant Int_Array := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
43         Sum : Integer := 0;
44     begin
45         for I in Arr'Range loop
46             Sum := Sum + Arr (I);
47         end loop;
```

(continues on next page)

(continued from previous page)

```

48     Ada.Text_IO.Put_Line ("Sum = " & Integer'Image (Sum));
49     end;
50 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loops_Ada
MD5: c09a092f8d2f682ce758d4bf059b954a

Runtime output

```

V = 128
V = 256
V = 30
V = 10
Sum = 55
```

The loop syntax in Ada is pretty straightforward. The **loop** and **end loop** keywords are used to open and close the loop scope. Instead of using the **break** keyword to exit the loop, Ada has the **exit** statement. The **exit** statement can be combined with a logic expression using the **exit when** syntax.

The major deviation in loop syntax is regarding for loops. You'll notice, in C, that you sometimes declare, and at least initialize a loop counter variable, specify a loop predicate, or an expression that indicates when the loop should continue executing or complete, and last you specify an expression to update the loop counter.

[C]

```

for (initialization expression; loop predicate; update expression) {
    // some statements
}
```

In Ada, you don't declare or initialize a loop counter or specify an update expression. You only name the loop counter and give it a range to loop over. The loop counter is **read-only**! You cannot modify the loop counter inside the loop like you can in C. And the loop counter will increment consecutively along the specified range. But what if you want to loop over the range in reverse order?

[C]

Listing 2100: main.c

```

1  #include <stdio.h>
2
3  #define MY_RANGE (10)
4
5  int main(int argc, const char * argv[])
6  {
7
8      for (int i = MY_RANGE; i >= 0; --i) {
9          printf("%d\n", i);
10     }
11
12     return 0;
13 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Counter_C
MD5: 4e70078ae51d113b8fa02340258c5ed5

Runtime output

```
10
9
8
7
6
5
4
3
2
1
0
```

[Ada]

Listing 2101: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5     My_Range : constant := 10;
6 begin
7     for I in reverse 0 .. My_Range loop
8         Put_Line (I'Img);
9     end loop;
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Counter_Ada
MD5: f25ed1a91c82620f16cd3084a6a0f475

Runtime output

```
10
9
8
7
6
5
4
3
2
1
0
```

Tick Image

Strangely enough, Ada people call the single apostrophe symbol, `'`, "tick". This "tick" says that we are accessing an attribute of the variable. When we do `'Img` on a variable of a numerical type, we are going to return the string version of that numerical type. So in the for loop above, `I'Img`, or "I tick image" will return the string representation of the numerical value stored in `I`. We have to do this because `Put_Line` is expecting a string as an input parameter.

We'll discuss attributes in more details [later in this chapter](#) (page 1782).

In the above example, we are traversing over the range in reverse order. In Ada, we use the **reverse** keyword to accomplish this.

In many cases, when we are writing a for loop, it has something to do with traversing an array. In C, this is a classic location for off-by-one errors. Let's see an example in action:

[C]

Listing 2102: main.c

```

1  #include <stdio.h>
2
3  #define LIST_LENGTH (100)
4
5  int main(int argc, const char * argv[])
6  {
7      int list[LIST_LENGTH];
8
9      for(int i = LIST_LENGTH; i > 0; --i) {
10         list[i] = LIST_LENGTH - i;
11     }
12
13     for (int i = 0; i < LIST_LENGTH; ++i)
14     {
15         printf("%d ", list[i]);
16
17         if (i % 10 == 0) {
18             printf("\n");
19         }
20     }
21
22     return 0;
23 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Reverse_C
MD5: 710ce30066551d1aada8d4e98a6004b1

Runtime output

```

0
99 98 97 96 95 94 93 92 91 90
89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60
59 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10
9 8 7 6 5 4 3 2 1
```

[Ada]

Listing 2103: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main
4  is
5      type Int_Array is array (Natural range 1 .. 100) of Integer;
6
7      List : Int_Array;
8  begin
9
```

(continues on next page)

(continued from previous page)

```

10  for I in reverse List'Range loop
11      List (I) := List'Last - I;
12  end loop;
13
14  for I in List'Range loop
15      Put (List (I)'Img & " ");
16
17      if I mod 10 = 0 then
18          New_Line;
19      end if;
20  end loop;
21
22  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Reverse_Ada
MD5: 340b935d42a80671bb050bdad1b032f7

Runtime output

```

99 98 97 96 95 94 93 92 91 90
89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60
59 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10
9  8  7  6  5  4  3  2  1  0

```

The above Ada and C code should initialize an array using a for loop. The initial values in the array should be contiguously decreasing from 99 to 0 as we index from the first index to the last index. In other words, the first index has a value of 99, the next has 98, the next 97 ... the last has a value of 0.

If you run both the C and Ada code above you'll notice that the outputs of the two programs are different. Can you spot why?

In the C code there are two problems:

1. There's a buffer overflow in the first iteration of the loop. We would need to modify the loop initialization to `int i = LIST_LENGTH - 1;`. The loop predicate should be modified to `i >= 0;`
2. The C code also has another off-by-one problem in the math to compute the value stored in `list[i]`. The expression should be changed to be `list[i] = LIST_LENGTH - i - 1;`.

These are typical off-by-one problems that plagues C programs. You'll notice that we didn't have this problem with the Ada code because we aren't defining the loop with arbitrary numeric literals. Instead we are accessing attributes of the array we want to manipulate and are using a keyword to determine the indexing direction.

We can actually simplify the Ada for loop a little further using iterators:

[Ada]

Listing 2104: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```

3 procedure Main
4 is
5     type Int_Array is array (Natural range 1 .. 100) of Integer;
6
7     List : Int_Array;
8 begin
9
10    for I in reverse List'Range loop
11        List (I) := List'Last - I;
12    end loop;
13
14    for I of List loop
15        Put (I'Img & " ");
16
17        if I mod 10 = 0 then
18            New_Line;
19        end if;
20    end loop;
21
22 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Reverse_Ada_Simplified
MD5: 612046826199b00ed61271d6215596fe

Runtime output

```

99 98 97 96 95 94 93 92 91 90
89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60
59 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10
9  8  7  6  5  4  3  2  1  0

```

In the second for loop, we changed the syntax to **for I of List**. Instead of I being the index counter, it is now an iterator that references the underlying element. This example of Ada code is identical to the last bit of Ada code. We just used a different method to index over the second for loop. There is no C equivalent to this Ada feature, but it is similar to C++'s range based for loop.

82.11 Type System

82.11.1 Strong Typing

Ada is considered a "strongly typed" language. This means that the language does not define any implicit type conversions. C does define implicit type conversions, sometimes referred to as *integer promotion*. The rules for promotion are fairly straightforward in simple expressions but can get confusing very quickly. Let's look at a typical place of confusion with implicit type conversion:

[C]

Listing 2105: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     unsigned char a = 0xFF;
6     char b = 0xFF;
7
8     printf("Does a == b?\n");
9     if(a == b)
10         printf("Yes.\n");
11     else
12         printf("No.\n");
13
14     printf("a: 0x%08X, b: 0x%08X\n", a, b);
15
16     return 0;
17 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_C
MD5: cab1ac9e2c86076d8435d53904783ba0

Runtime output

```
Does a == b?
No.
a: 0x000000FF, b: 0xFFFFFFFF
```

Run the above code. You will notice that `a != b`! If we look at the output of the last `printf` statement we will see the problem. `a` is an unsigned number where `b` is a signed number. We stored a value of `0xFF` in both variables, but `a` treated this as the decimal number `255` while `b` treated this as the decimal number `-1`. When we compare the two variables, of course they aren't equal; but that's not very intuitive. Let's look at the equivalent Ada example:

[Ada]

Listing 2106: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5     type Char is range 0 .. 255;
6     type Unsigned_Char is mod 256;
7
8     A : Char := 16#FF#;
9     B : Unsigned_Char := 16#FF#;
10 begin
11
12     Put_Line ("Does A = B?");
13
14     if A = B then
15         Put_Line ("Yes");
16     else
17         Put_Line ("No");
18     end if;
19
20 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_Ada
 MD5: d6ef2668809159e9fb0d42f91e893222

Build output

```
main.adb:14:09: error: invalid operand types for operator "="
main.adb:14:09: error: left operand has type "Char" defined at line 5
main.adb:14:09: error: right operand has type "Unsigned_Char" defined at line 6
gprbuild: *** compilation phase failed
```

If you try to run this Ada example you will get a compilation error. This is because the compiler is telling you that you cannot compare variables of two different types. We would need to explicitly cast one side to make the comparison against two variables of the same type. By enforcing the explicit cast we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect.

Another example: you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

Listing 2107: strong_typing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Strong_Typing is
4   Alpha : constant Integer := 1;
5   Beta  : constant Integer := 10;
6   Result : Float;
7 begin
8   Result := Float (Alpha) / Float (Beta);
9
10  Put_Line (Float'Image (Result));
11 end Strong_Typing;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_Ada_2
 MD5: bf91f01b499bcd7da1df751a9f91a767

Runtime output

```
1.00000E-01
```

[C]

Listing 2108: main.c

```
1 #include <stdio.h>
2
3 void weakTyping (void) {
4   const int alpha = 1;
5   const int beta = 10;
6   float result;
7
8   result = alpha / beta;
```

(continues on next page)

(continued from previous page)

```
9
10     printf("%f\n", result);
11 }
12
13 int main(int argc, const char * argv[])
14 {
15     weakTyping();
16
17     return 0;
18 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_C_2
MD5: e4310900cd1195d6e3d349e0c4aa758a

Runtime output

0.000000

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from **Integer** to **Float** explicit. In fact, it significantly changes the behavior of the computation. While the Ada code performs a floating point operation $1.0 / 10.0$ and stores 0.1 in Result, the C version instead store 0.0 in result. This is because the C version perform an integer operation between two integer variables: $1 / 10$ is 0. The result of the integer division is then converted to a **float** and stored. Errors of this sort can be very hard to locate in complex pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to **Float**:

[Ada]

```
-- Perform an Integer division then convert to Float
Result := Float (Alpha / Beta);
```

The complete example would then be:

[Ada]

Listing 2109: strong_typing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Strong_Typing is
4     Alpha : constant Integer := 1;
5     Beta  : constant Integer := 10;
6     Result : Float;
7 begin
8     Result := Float (Alpha / Beta);
9
10    Put_Line (Float'Image (Result));
11 end Strong_Typing;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Strong_Typing_Ada_2
MD5: 50d6a6a3270b51880c43c07f077760b6

Runtime output

```
0.00000E+00
```

i Floating Point Literals

In Ada, a floating point literal must be written with both an integral and decimal part. `10` is not a valid literal for a floating point value, while `10.0` is.

82.11.2 Language-Defined Types

The principal scalar types predefined by Ada are **Integer**, **Float**, **Boolean**, and **Character**. These correspond to **int**, **float**, **int** (when used for Booleans), and **char**, respectively. The names for these types are not reserved words; they are regular identifiers. There are other language-defined integer and floating-point types as well. All have implementation-defined ranges and precision.

82.11.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators `+`, `-`, `*`, and `/` for integer and floating-point types take operands of the same type and return a value of that type.

[Ada]

Listing 2110: main.adb

```

1  procedure Main is
2      type Distance is new Float;
3      type Area is new Float;
4
5      D1 : Distance := 2.0;
6      D2 : Distance := 3.0;
7      A  : Area;
8  begin
9      D1 := D1 + D2; -- OK
10     D1 := D1 + A;  -- NOT OK: incompatible types for "+"
11     A  := D1 * D2; -- NOT OK: incompatible types for ":="
12     A  := Area (D1 * D2); -- OK
13 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Application_Defined_Types
MD5: 6a21d6281cc529bbf8ce2216d7e4a770

Build output

```
main.adb:10:13: error: invalid operand types for operator "+"
main.adb:10:13: error: left operand has type "Distance" defined at line 2
main.adb:10:13: error: right operand has type "Area" defined at line 3
```

(continues on next page)

(continued from previous page)

```
main.adb:11:13: error: expected type "Area" defined at line 3
main.adb:11:13: error: found type "Distance" defined at line 2
gprbuild: *** compilation phase failed
```

Even though the Distance and Area types above are just **Float**, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two Distance yields a Distance) and prohibit some useful cases (for example multiplying two Distances should deliver an Area). These situations can be handled through other mechanisms. A predefined operation can be identified as abstract to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C **enum**:

[Ada]

Listing 2111: main.adb

```
1  procedure Main is
2      type Day is
3          (Monday,
4           Tuesday,
5           Wednesday,
6           Thursday,
7           Friday,
8           Saturday,
9           Sunday);
10
11     D : Day := Monday;
12 begin
13     null;
14 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_Ada
MD5: 51abd1863970e14ff86859c1aae11fe8

[C]

Listing 2112: main.c

```
1  enum Day {
2      Monday,
3      Tuesday,
4      Wednesday,
5      Thursday,
6      Friday,
7      Saturday,
8      Sunday
9  };
10
11 int main(int argc, const char * argv[])
12 {
13     enum Day d = Monday;
14 }
```

(continues on next page)

(continued from previous page)

```

15     return 0;
16 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_C
MD5: d9f6724759375a126a6b5d8dceea3f24

But even though such enumerations may be implemented by the compiler as numeric values, at the language level Ada will not confuse the fact that Monday is a Day and is not an **Integer**. You can compare a Day with another Day, though. To specify implementation details like the numeric values that correspond with enumeration values in C you include them in the original **enum** declaration:

[C]

Listing 2113: main.c

```

1  #include <stdio.h>
2
3  enum Day {
4      Monday    = 10,
5      Tuesday   = 11,
6      Wednesday = 12,
7      Thursday  = 13,
8      Friday    = 14,
9      Saturday  = 15,
10     Sunday    = 16
11 };
12
13 int main(int argc, const char * argv[])
14 {
15     enum Day d = Monday;
16
17     printf("d = %d\n", d);
18
19     return 0;
20 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_Values_C
MD5: 48ae1c84dafabde7a16de5305e106a80

Runtime output

```
d = 10
```

But in Ada you must use both a type definition for Day as well as a separate representation clause for it like:

[Ada]

Listing 2114: main.adb

```

1  with Ada.Text_IO;
2
3  procedure Main is
4      type Day is
5          (Monday,
6           Tuesday,

```

(continues on next page)

(continued from previous page)

```

7      Wednesday,
8      Thursday,
9      Friday,
10     Saturday,
11     Sunday);
12
13     -- Representation clause for Day type:
14     for Day use
15         (Monday    => 10,
16          Tuesday   => 11,
17          Wednesday => 12,
18          Thursday  => 13,
19          Friday    => 14,
20          Saturday  => 15,
21          Sunday    => 16);
22
23     D : Day := Monday;
24     V : Integer;
25 begin
26     V := Day'Enum_Rep (D);
27     Ada.Text_IO.Put_Line (Integer'Image (V));
28 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enumeration_Values
MD5: 9a4fala899cb8c240105bf8ad6dbfde3

Runtime output

10

Note that however, unlike C, values for enumerations in Ada have to be unique.

82.11.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the **range** reserved word, for example:

[Ada]

Listing 2115: main.adb

```

1  procedure Main is
2      type Grade is range 0 .. 100;
3
4      G1, G2 : Grade;
5      N      : Integer;
6  begin
7      -- ...           -- Initialization of N
8      G1 := 80;        -- OK
9      G1 := N;         -- Illegal (type mismatch)
10     G1 := Grade (N);  -- Legal, run-time range check
11     G2 := G1 + 10;    -- Legal, run-time range check
12     G1 := (G1 + G2) / 2; -- Legal, run-time range check
13 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Range_Check
 MD5: 0f249b06e373497ae94b6055a37187c8

Build output

```
main.adb:9:10: error: expected type "Grade" defined at line 2
main.adb:9:10: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed
```

In the above example, Grade is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to G1 is correct and will not raise a run-time exception. Assigning N to G1 is illegal since Grade is a different type than **Integer**. Converting N to Grade makes the assignment legal, and a range check on the conversion confirms that the value is within 0 .. 100. Assigning G1 + 10 to G2 is legal since + for Grade returns a Grade (note that the literal 10 is interpreted as a Grade value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression G1 + G2 may be outside the range of Grade, but the final result will be in range. Nevertheless, depending on the representation chosen for Grade, the addition may overflow. If the compiler represents Grade values as signed 8-bit integers (i.e., machine numbers in the range -128 .. 127) then the sum G1 + G2 may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

[Ada]

Listing 2116: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4   type Grade is range 0 .. 100;
5
6   G1, G2 : Grade := 99;
7 begin
8   G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
9   Ada.Text_IO.Put_Line (Grade'Image (G1));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Range_And_Explicit_Conversion
 MD5: d317fd95099e49017c4a4c1c52b7f8be

Runtime output

99

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the Day example, one can write:

[Ada]

Listing 2117: main.adb

```
1 procedure Main is
2   type Day is
3     (Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday);
10
11   type Business_Day is new Day range Monday .. Friday;
12   type Weekend_Day is new Day range Saturday .. Sunday;
13 begin
14   null;
15 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enum_Ranges_1
MD5: fd775ad4990d5636607d3a0d9b00044d

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind *subtypes* in Ada. A subtype is a type with optional additional constraints. For example:

[Ada]

Listing 2118: main.adb

```
1 procedure Main is
2   type Day is
3     (Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday);
10
11   subtype Business_Day is Day range Monday .. Friday;
12   subtype Weekend_Day is Day range Saturday .. Sunday;
13   subtype Dice_Throw is Integer range 1 .. 6;
14 begin
15   null;
16 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Enum_Ranges_2
MD5: 5bcbde5b9f1aea57ff172fcfc89e1c41

These declarations don't create new types, just new names for constrained ranges of their base types.

The purpose of numeric ranges is to express some application-specific constraint that we want the compiler to help us enforce. More importantly, we want the compiler to tell us when that constraint cannot be met — when the underlying hardware cannot support the range given. There are two things to consider:

- just a range constraint, such as A : **Integer range 0 .. 10**;;, or

- a type declaration, such as `type Result is range 0 .. 1_000_000_000;`.

Both represent some sort of application-specific constraint, but in addition, the type declaration promotes portability because it won't compile on targets that do not have a sufficiently large hardware numeric type. That's a definition of portability that is preferable to having something compile anywhere but not run correctly, as in C.

82.11.5 Unsigned And Modular Types

Unsigned integer numbers are quite common in embedded applications. In C, you can use them by declaring `unsigned int` variables. In Ada, you have two options:

- declare custom *unsigned* range types;
 - In addition, you can declare custom range *subtypes* or use existing subtypes such as `Natural`.
- declare custom modular types.

The following table presents the main features of each type. We discuss these types right after.

Feature	[C] <code>unsigned int</code>	[Ada] Unsigned range	[Ada] Modular
Excludes negative value	✓	✓	✓
Wraparound	✓		✓

When declaring custom range types in Ada, you may use the full range in the same way as in C. For example, this is the declaration of a 32-bit unsigned integer type and the X variable in Ada:

[Ada]

Listing 2119: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
5
6   X : Unsigned_Int_32 := 42;
7 begin
8   Put_Line ("X = " & Unsigned_Int_32'Image (X));
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unsigned_32_Ada
MD5: 0a179ce327c022468f66b6814a981b62

Runtime output

X = 42

In C, when `unsigned int` has a size of 32 bits, this corresponds to the following declaration:
[C]

Listing 2120: main.c

```
1 #include <stdio.h>
2 #include <limits.h>
```

(continues on next page)

(continued from previous page)

```
3
4 int main(int argc, const char * argv[])
5 {
6     unsigned int x = 42;
7     printf("x = %u\n", x);
8
9     return 0;
10 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unsigned_32_C
MD5: 546068de216de96282490e81a0f7df26

Runtime output

x = 42

Another strategy is to declare subtypes for existing signed types and specify just the range that excludes negative numbers. For example, let's declare a custom 32-bit signed type and its unsigned subtype:

[Ada]

Listing 2121: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     type Signed_Int_32 is range -2 ** 31 .. 2 ** 31 - 1;
5
6     subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. Signed_Int_32'Last;
7     -- Equivalent to:
8     -- subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. 2 ** 31 - 1;
9
10    X : Unsigned_Int_31 := 42;
11 begin
12    Put_Line ("X = " & Unsigned_Int_31'Image (X));
13 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unsigned_31_Ada
MD5: 2ef2b5bfd54821ceb35faa222e649156

Runtime output

X = 42

In this case, we're just skipping the sign bit of the Signed_Int_32 type. In other words, while Signed_Int_32 has a size of 32 bits, Unsigned_Int_31 has a range of 31 bits, even if the base type has 32 bits.

Note that the declaration above is actually similar to the existing **Natural** subtype. Ada provides the following standard subtypes:

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Since they're standard subtypes, you can declare variables of those subtypes directly in your implementation, in the same way as you can declare **Integer** variables.

As indicated in the table above, however, there is a difference in behavior for the variables we just declared, which occurs in case of overflow. Let's consider this C example:

[C]

Listing 2122: main.c

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(int argc, const char * argv[])
5  {
6      unsigned int x = UINT_MAX + 1;
7      /* Now: x == 0 */
8
9      printf("x = %u\n", x);
10
11     return 0;
12 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overflow_Wraparound_C
MD5: 7d5dcf65471304ff8f303195359b4790

Runtime output

x = 0

The corresponding code in Ada raises an exception:

[Ada]

Listing 2123: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
5
6      X : Unsigned_Int_32 := Unsigned_Int_32'Last + 1;
7      -- Overflow: exception is raised!
8  begin
9      Put_Line ("X = " & Unsigned_Int_32'Image (X));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overflow_Wraparound_Ada
MD5: ee4c3e905c59f5c8d87311e13d079836

Build output

main.adb:6:48: warning: value not in range of type "Unsigned_Int_32" defined at ↵
↵line 4 [enabled by default]
main.adb:6:48: warning: Constraint_Error will be raised at run time [enabled by ↵
↵default]

Runtime output

raised CONSTRAINT_ERROR : main.adb:6 range check failed

While the C uses modulo arithmetic for unsigned integer, Ada doesn't use it for the `Unsigned_Int_32` type. Ada does, however, support modular types via type definitions using the `mod` keyword. In this example, we declare a 32-bit modular type:

[Ada]

Listing 2124: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Unsigned_32 is mod 2**32;
5
6   X : Unsigned_32 := Unsigned_32'Last + 1;
7   -- Now: X = 0
8 begin
9   Put_Line ("X = " & Unsigned_32'Image (X));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overflow_Wraparound_Ada
MD5: 4ed963ab372cafc8e7a19d9c3107276b

Runtime output

X = 0

In this case, the behavior is the same as in the C declaration above.

Modular types, unlike Ada's signed integers, also provide bit-wise operations, a typical application for unsigned integers in C. In Ada, you can use operators such as `and`, `or`, `xor` and `not`. You can also use typical bit-shifting operations, such as `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`.

82.11.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The `'Image` and `'Value` attributes allow you to transform a scalar value into a `String` and vice-versa. For example:

[Ada]

Listing 2125: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   A : Integer := 10;
5 begin
6   Put_Line (Integer'Image (A));
7   A := Integer'Value ("99");
8   Put_Line (Integer'Image (A));
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Image_Attribute
 MD5: 1fcfc79ec599a26e21aef7eacffcf96e

Runtime output

10
99

Important

Semantically, attributes are equivalent to subprograms. For example, `Integer'Image` is defined as follows:

```
function Integer'Image(Arg : Integer'Base) return String;
```

Certain attributes are provided only for certain kinds of types. For example, the `'Val` and `'Pos` attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

Listing 2126: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   C : Character := 'a';
5 begin
6   Put (C);
7   C := Character'Val (Character'Pos (C) + 1);
8   Put (C);
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Character_1
 MD5: 742bbaeb74e5dd9fa73089c0d1aa0fde

Runtime output

ab

A more concise way to get the next value in Ada is to use the `'Succ` attribute:

[Ada]

Listing 2127: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   C : Character := 'a';
5 begin
6   Put (C);
7   C := Character'Succ (C);
8   Put (C);
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Character_1
MD5: 842eeff2b82dcdb8c73547a33d03995b

Runtime output

ab

You can get the previous value using the '[Pred](#)' attribute. Here is the equivalent in C:
[C]

Listing 2128: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     char c = 'a';
6     printf("%c", c);
7     c++;
8     printf("%c", c);
9
10    return 0;
11 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Loop_Reverse_C
MD5: 40bfbd6a672bc3fdb7e8f2f2d7101b19

Runtime output

ab

Other interesting examples are the '[First](#)' and '[Last](#)' attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, **Integer**'[First](#)' returns -2^{31} and **Integer**'[Last](#)' returns $2^{31} - 1$.

82.11.7 Arrays and Strings

C arrays are pointers with offsets, but the same is not the case for Ada. Arrays in Ada are not interchangeable with operations on pointers, and array types are considered first-class citizens. They have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from '[a](#)' to '[z](#)':

[Ada]

Listing 2129: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     type Arr_Type is array (Integer range <>) of Character;
5     Arr : Arr_Type (1 .. 26);
```

(continues on next page)

(continued from previous page)

```

6   C : Character := 'a';
7   begin
8     for I in Arr'Range loop
9       Arr (I) := C;
10      C := Character'Succ (C);
11
12      Put (Arr (I) & " ");
13
14      if I mod 7 = 0 then
15        New_Line;
16      end if;
17    end loop;
18  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Range_Ada
MD5: 8e0597f6c040c740b35c79bc4706829b

Runtime output

```

a b c d e f g
h i j k l m n
o p q r s t u
v w x y z

```

[C]

Listing 2130: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      char Arr [26];
6      char C = 'a';
7
8      for (int I = 0; I < 26; ++I) {
9          Arr [I] = C++;
10         printf ("%C ", Arr [I]);
11
12         if ((I + 1) % 7 == 0) {
13             printf ("\n");
14         }
15     }
16
17     return 0;
18 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Range_C
MD5: 1182155f46a0b69f73cd5937c23ed67d

Runtime output

```

a b c d e f g
h i j k l m n
o p q r s t u
v w x y z

```

In C, only the size of the array is given during declaration. In Ada, array index ranges are

specified using two values of a discrete type. In this example, the array type declaration specifies the use of **Integer** as the index type, but does not provide any constraints (use `<>`, pronounced *box*, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called `'Range'`. In our example, `Arr'Range` can also be expressed as `Arr'First .. Arr'Last`; both expressions will resolve to `1 .. 26`. So the `'Range` attribute supplies the bounds for our **for** loop. There is no risk of stating either of the bounds incorrectly, as one might do in C where `I <= 26` may be specified as the end-of-loop condition.

As in C, Ada **String** is an array of **Character**. Ada strings, importantly, are not delimited with the special character `'0'` like they are in C. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined **String** type is very straightforward to use:

[Ada]

Listing 2131: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   My_String : String (1 .. 19) := "This is an example!";
5 begin
6   Put_Line (My_String);
7 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Constrained_String
MD5: da2e88900c670f80b7380f87f2b89ec2

Runtime output

This is an example!

Unlike C, Ada does not offer escape sequences such as `'n'`. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, `&`). Here for example, is how to initialize a line of text ending with a new line:

[Ada]

Listing 2132: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   My_String : String := "This is a line" & ASCII.LF;
5 begin
6   Put (My_String);
7 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Constrained_String
MD5: 684bbdbf99d48ed6fd5c257183a6609f

Runtime output

This is a line

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of `My_String`'s bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

Listing 2133: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Arr_Type is array (Integer range <>) of Integer;
5      A1 : Arr_Type (1 .. 2);
6      A2 : Arr_Type (1 .. 2);
7  begin
8      A1 (1) := 0;
9      A1 (2) := 1;
10
11     A2 := A1;
12
13     for I in A2'Range loop
14         Put_Line (Integer'Image (A2 (I)));
15     end loop;
16 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Copy_Ada
MD5: 4d4e9aa063c1f488e7cefa90083d06c2

Runtime output

```

0
1

```

[C]

Listing 2134: main.c

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, const char * argv[])
5  {
6      int A1 [2];
7      int A2 [2];
8
9      A1 [0] = 0;
10     A1 [1] = 1;
11
12     memcpy (A2, A1, sizeof (int) * 2);
13
14     for (int i = 0; i < 2; i++) {
15         printf("%d\n", A2[i]);
16     }
17
18     return 0;
19 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Copy_C
MD5: 0dade800452673b7a82afe1c656f07e6

Runtime output

```
0
1
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

Listing 2135: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr_Type is array (Integer range <>) of Integer;
5   A1 : Arr_Type (1 .. 10) := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
6   A2 : Arr_Type (1 .. 5)  := (1, 2, 3, 4, 5);
7 begin
8   A2 (1 .. 3) := A1 (4 .. 6);
9
10  for I in A2'Range loop
11    Put_Line (Integer'Image (A2 (I)));
12  end loop;
13 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Slice
MD5: cb2a7de2cff8ea19025363886f8821e4

Runtime output

```
4
5
6
4
5
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

Listing 2136: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr_Type is array (Integer range <>) of Integer;
5   A1 : Arr_Type (1 .. 2) := (10, 20);
6   A2 : Arr_Type (1 .. 2) := (10, 20);
7 begin
8   if A1 = A2 then
```

(continues on next page)

(continued from previous page)

```

9      Put_Line ("A1 = A2");
10     else
11       Put_Line ("A1 /= A2");
12     end if;
13 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Equal_Ada
MD5: 650a734875a02b2fb3678bbc3f8dd82a

Runtime output

A1 = A2

[C]

Listing 2137: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int A1 [2] = { 10, 20 };
6      int A2 [2] = { 10, 20 };
7
8      int eq = 1;
9
10     for (int i = 0; i < 2; ++i) {
11         if (A1 [i] != A2 [i]) {
12             eq = 0;
13             break;
14         }
15     }
16
17     if (eq) {
18         printf("A1 == A2\n");
19     }
20     else {
21         printf("A1 != A2\n");
22     }
23
24     return 0;
25 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Equal_C
MD5: efe8717d931324bcbe8b70b03693c92e

Runtime output

A1 == A2

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the others expression to define a default value for all the unspecified array elements. Therefore, you can write:

[Ada]

Listing 2138: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr_Type is array (Integer range <>) of Integer;
5   A1 : Arr_Type (-2 .. 42) := (others => 0);
6 begin
7   -- use a slice to assign A1 elements 11 .. 19 to 1
8   A1 (11 .. 19) := (others => 1);
9
10  Put_Line ("---- A1 ----");
11  for I in A1'Range loop
12    Put_Line (Integer'Image (I) & " => " &
13              Integer'Image (A1 (I)));
14  end loop;
15 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Assignment_Ada
MD5: 673d31f633a32b6bb1cce238150cfc80

Runtime output

```
---- A1 ----
-2 => 0
-1 => 0
0 => 0
1 => 0
2 => 0
3 => 0
4 => 0
5 => 0
6 => 0
7 => 0
8 => 0
9 => 0
10 => 0
11 => 1
12 => 1
13 => 1
14 => 1
15 => 1
16 => 1
17 => 1
18 => 1
19 => 1
20 => 0
21 => 0
22 => 0
23 => 0
24 => 0
25 => 0
26 => 0
27 => 0
28 => 0
29 => 0
30 => 0
31 => 0
32 => 0
33 => 0
```

(continues on next page)

(continued from previous page)

```

34 => 0
35 => 0
36 => 0
37 => 0
38 => 0
39 => 0
40 => 0
41 => 0
42 => 0

```

In this example, we're specifying that A1 has a range between -2 and 42. We use (**others** => 0) to initialize all array elements with zero. In the next example, the number of elements is determined by looking at the right-hand side:

[Ada]

Listing 2139: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr_Type is array (Integer range <>) of Integer;
5   A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
6 begin
7   A1 := (1, 2, 3, others => 10);
8
9   Put_Line ("---- A1 ----");
10  for I in A1'Range loop
11    Put_Line (Integer'Image (I) & " => " &
12              Integer'Image (A1 (I)));
13  end loop;
14 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Array_Assignment_Ada
MD5: 3e3d69815373d1c61208df265903e89d

Runtime output

```

---- A1 ----
-2147483648 => 1
-2147483647 => 2
-2147483646 => 3
-2147483645 => 10
-2147483644 => 10
-2147483643 => 10
-2147483642 => 10
-2147483641 => 10
-2147483640 => 10

```

Since A1 is initialized with an aggregate of 9 elements, A1 automatically has 9 elements. Also, we're not specifying any range in the declaration of A1. Therefore, the compiler uses the default range of the underlying array type Arr_Type, which has an unconstrained range based on the **Integer** type. The compiler selects the first element of that type (**Integer'First**) as the start index of A1. If you replaced **Integer range <>** in the declaration of the Arr_Type by **Positive range <>**, then A1's start index would be **Positive'First** — which corresponds to one.

82.11.8 Heterogeneous Data Structures

The structure corresponding to a C **struct** is an Ada **record**. Here are some simple records:
[Ada]

Listing 2140: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type R is record
5     A, B : Integer;
6     C    : Float;
7   end record;
8
9   V : R;
10 begin
11   V.A := 0;
12   Put_Line ("V.A = " & Integer'Image (V.A));
13 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Struct_Ada
MD5: 013f27dfc827355f32bea37fb267df9b

Runtime output

V.A = 0

[C]

Listing 2141: main.c

```
1 #include <stdio.h>
2
3 struct R {
4     int A, B;
5     float C;
6 };
7
8 int main(int argc, const char * argv[])
9 {
10     struct R V;
11     V.A = 0;
12     printf("V.A = %d\n", V.A);
13
14     return 0;
15 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Struct_C
MD5: 653b65bbb6ea02a512e439d912e11d7f

Runtime output

V.A = 0

Ada allows specification of default values for fields just like C. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by others => <> to specify that fields not listed will take their default values. For example:

[Ada]

Listing 2142: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      type R is record
6          A, B : Integer := 0;
7          C   : Float   := 0.0;
8      end record;
9
10     procedure Put_R (V : R; Name : String) is
11     begin
12         Put_Line (Name & " = ("
13                 & Integer'Image (V.A) & ", "
14                 & Integer'Image (V.B) & ", "
15                 & Float'Image (V.C) & ")");
16     end Put_R;
17
18     V1 : constant R := (1, 2, 1.0);
19     V2 : constant R := (A => 1, B => 2, C => 1.0);
20     V3 : constant R := (C => 1.0, A => 1, B => 2);
21     V4 : constant R := (C => 1.0, others => <>);
22
23 begin
24     Put_R (V1, "V1");
25     Put_R (V2, "V2");
26     Put_R (V3, "V3");
27     Put_R (V4, "V4");
28 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Struct_Default_Ada
MD5: d0a9713e3bd9804c00ebf68cc7c196b7

Runtime output

```

V1 = ( 1,  2,  1.00000E+00)
V2 = ( 1,  2,  1.00000E+00)
V3 = ( 1,  2,  1.00000E+00)
V4 = ( 0,  0,  1.00000E+00)

```

82.11.9 Pointers

As a foreword to the topic of pointers, it's important to keep in mind the fact that most situations that would require a pointer in C do not in Ada. In the vast majority of cases, indirect memory management can be hidden from the developer and thus saves from many potential errors. However, there are situation that do require the use of pointers, or said differently that require to make memory indirection explicit. This section will present Ada access types, the equivalent of C pointers. A further section will provide more details as to how situations that require pointers in C can be done without access types in Ada.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

Listing 2143: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type R is record
5          A, B : Integer;
6      end record;
7
8      procedure Put_R (V : R; Name : String) is
9      begin
10         Put_Line (Name & " = ("
11                     & Integer'Image (V.A) & ", "
12                     & Integer'Image (V.B) & ")");
13     end Put_R;
14
15     V1, V2 : R;
16
17     begin
18         V1.A := 0;
19         V1.B := 0;
20         V2 := V1;
21         V2.A := 1;
22
23         Put_R (V1, "V1");
24         Put_R (V2, "V2");
25     end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Pointers_Ada
MD5: 0a4ade5aa12de5b0760d149c8e319d5f

Runtime output

```

V1 = ( 0, 0)
V2 = ( 1, 0)

```

[C]

Listing 2144: main.c

```

1  #include <stdio.h>
2
3  struct R {
4      int A, B;
5  };
6
7  void print_r(const struct R *v,
8              const char *name)
9  {
10     printf("%s = (%d, %d)\n", name, v->A, v->B);
11 }
12
13 int main(int argc, const char * argv[])
14 {
15     struct R V1, V2;
16     V1.A = 0;
17     V1.B = 0;
18     V2 = V1;
19     V2.A = 1;
20

```

(continues on next page)

(continued from previous page)

```

21     print_r(&V1, "V1");
22     print_r(&V2, "V2");
23
24     return 0;
25 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Pointers_C
MD5: e604bf19c98927151f3b4989dc50747a

Runtime output

```

V1 = (0, 0)
V2 = (1, 0)

```

There are many commonalities between the Ada and C semantics above. In Ada and C, objects are allocated on the stack and are directly accessed. V1 and V2 are two different objects and the assignment statement copies the value of V1 into V2. V1 and V2 are two distinct objects.

Here's now a similar example, but using heap allocation instead:

[Ada]

Listing 2145: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type R is record
5          A, B : Integer;
6      end record;
7
8      type R_Access is access R;
9
10     procedure Put_R (V : R; Name : String) is
11     begin
12         Put_Line (Name & " = ("
13                 & Integer'Image (V.A) & ", "
14                 & Integer'Image (V.B) & ")");
15     end Put_R;
16
17     V1 : R_Access;
18     V2 : R_Access;
19 begin
20     V1 := new R;
21     V1.A := 0;
22     V2 := V1;
23     V2.A := 1;
24
25     Put_R (V1.all, "V1");
26     Put_R (V2.all, "V2");
27 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Heap_Alloc_Ada
MD5: 963b48bb0a8585a9941d8fb2d0eda390

Runtime output

```
V1 = ( 1, 0)
V2 = ( 1, 0)
```

[C]

Listing 2146: main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct R {
5      int A, B;
6  };
7
8  void print_r(const struct R *v,
9              const char *name)
10 {
11     printf("%s = (%d, %d)\n", name, v->A, v->B);
12 }
13
14 int main(int argc, const char * argv[])
15 {
16     struct R * V1, * V2;
17     V1 = malloc(sizeof(struct R));
18     V1->A = 0;
19     V2 = V1;
20     V2->A = 1;
21
22     print_r(V1, "V1");
23     print_r(V2, "V2");
24
25     return 0;
26 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Heap_Alloc_C
MD5: 5c832377403dfa8f00d70ef92bfeff65

Runtime output

```
V1 = (1, 0)
V2 = (1, 0)
```

In this example, an object of type R is allocated on the heap. The same object is then referred to through V1 and V2. As in C, there's no garbage collector in Ada, so objects allocated by the new operator need to be expressly freed (which is not the case here).

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append **.all**. The equivalent of V1->A in C can be written either as V1.A or V1.all.A.

Pointers to scalar objects in Ada and C look like:

[Ada]

Listing 2147: main.adb

```
1  procedure Main is
2      type A_Int is access Integer;
3      Var : A_Int := new Integer;
```

(continues on next page)

(continued from previous page)

```

4 begin
5   Var.all := 0;
6 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_To_Scalars
MD5: 2e2bf53a9b5dc1098921d811be73a7f0

[C]

Listing 2148: main.c

```

1 #include <stdlib.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int * Var = malloc (sizeof(int));
6     *Var = 0;
7     return 0;
8 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Pointers_To_Scalars
MD5: f22d7b6f8170587009b0f6bb1299c0a0

In Ada, an initializer can be specified with the allocation by appending '(value)':

[Ada]

Listing 2149: main.adb

```

1 procedure Main is
2   type A_Int is access Integer;
3
4   Var : A_Int := new Integer'(0);
5 begin
6   null;
7 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_Initialization
MD5: 5789253068f77100eec34919b8de66ec

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being aliased. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either **access all** (if the referenced object needs to be assigned to) or **access constant** (if the referenced object is a constant). The '**Access**' attribute works like the C & operator to get a pointer to the object, but with a *scope accessibility* check to prevent references to objects that have gone out of scope. For example:

[Ada]

Listing 2150: main.adb

```

1 procedure Main is
2   type A_Int is access all Integer;
3   Var : aliased Integer;

```

(continues on next page)

(continued from previous page)

```
4   Ptr : A_Int := Var'Access;  
5 begin  
6   null;  
7 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_All
MD5: 520df34083e3517876e10710530380be

[C]

Listing 2151: main.c

```
1 int main(int argc, const char * argv())  
2 {  
3     int Var;  
4     int * Ptr = &Var;  
5  
6     return 0;  
7 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Access_All_C
MD5: a592fcf09dabe15f2aaf12fba047d74f

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs, it's called `Ada.Unchecked_Deallocation`. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

Listing 2152: main.adb

```
1 with Ada.Unchecked_Deallocation;  
2  
3 procedure Main is  
4     type Integer_Access is access all Integer;  
5     procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);  
6     My_Pointer : Integer_Access := new Integer;  
7 begin  
8     Free (My_Pointer);  
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Unchecked_Deallocation
MD5: ef6ee170fe1f6c6c01037a09809916f

[C]

Listing 2153: main.c

```
1 #include <stdlib.h>  
2  
3 int main(int argc, const char * argv())  
4 {  
5     int * my_pointer = malloc (sizeof(int));
```

(continues on next page)

(continued from previous page)

```

6   free (my_pointer);
7
8   return 0;
9 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Free
 MD5: 066046816dd1c4f9106b5e822cfe5e44

We'll discuss generics later *in this section* (page 1895).

82.12 Functions and Procedures

82.12.1 General Form

Subroutines in C are always expressed as functions which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term *subprogram* to refer to both functions and procedures.

Parameters can be passed in three distinct modes:

- **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram.
- **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller.
- **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant pointer in C).

Ada also provides **access** and **aliased** parameters, which are in effect explicit pass-by-reference indicators.

In Ada, the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). C has the programmer specify how to pass the parameter.

i Important

There are some exceptions to the "general" rule in Ada. For example, parameters of scalar types are always passed by copy, for all three modes.

Here's a first example:

[Ada]

Listing 2154: proc.ads

```

1  procedure Proc
2  (Var1 : Integer;
3   Var2 : out Integer;
4   Var3 : in out Integer);

```

Listing 2155: func.ads

```
1 function Func (Var : Integer) return Integer;
```

Listing 2156: proc.adb

```
1 with Func;  
2  
3 procedure Proc  
4   (Var1 : Integer;  
5    Var2 : out Integer;  
6    Var3 : in out Integer)  
7 is  
8 begin  
9   Var2 := Func (Var1);  
10  Var3 := Var3 + 1;  
11 end Proc;
```

Listing 2157: func.adb

```
1 function Func (Var : Integer) return Integer  
2 is  
3 begin  
4   return Var + 1;  
5 end Func;
```

Listing 2158: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2 with Proc;  
3  
4 procedure Main is  
5   V1, V2 : Integer;  
6 begin  
7   V2 := 2;  
8   Proc (5, V1, V2);  
9  
10  Put_Line ("V1: " & Integer'Image (V1));  
11  Put_Line ("V2: " & Integer'Image (V2));  
12 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Subroutines_Ada
MD5: a35fb6ae1b37325c3f39b3316e4246a8

Runtime output

```
V1: 6  
V2: 3
```

[C]

Listing 2159: proc.h

```
1 void Proc  
2   (int Var1,  
3    int * Var2,  
4    int * Var3);
```

Listing 2160: func.h

```
1 int Func (int Var);
```

Listing 2161: proc.c

```
1 #include "func.h"
2
3 void Proc
4     (int Var1,
5      int * Var2,
6      int * Var3)
7 {
8     *Var2 = Func (Var1);
9     *Var3 += 1;
10 }
```

Listing 2162: func.c

```
1 int Func (int Var)
2 {
3     return Var + 1;
4 }
```

Listing 2163: main.c

```
1 #include <stdio.h>
2 #include "proc.h"
3
4 int main(int argc, const char * argv[])
5 {
6     int v1, v2;
7
8     v2 = 2;
9     Proc (5, &v1, &v2);
10
11     printf("v1: %d\n", v1);
12     printf("v2: %d\n", v2);
13
14     return 0;
15 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Subroutines_C
MD5: dd5645c832ef00b94061f204852084a3

Runtime output

```
v1: 6
v2: 3
```

The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared.

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax (except for the modes), including de-

fault values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

In Ada 202X

Ada 202X allows for using static expression functions, which are evaluated at compile time. To achieve this, we can use an aspect — we'll discuss aspects *later in this chapter* (page 1804).

An expression function is static when the Static aspect is specified. For example:

```
procedure Main is
    X1 : constant := (if True then 37 else 42);

    function If_Then_Else (Flag : Boolean; X, Y : Integer)
        return Integer is
        (if Flag then X else Y) with Static;

    X2 : constant := If_Then_Else (True, 37, 42);

begin
    null;
end Main;
```

In this example, we declare X1 using an expression. In the declaration of X2, we call the static expression function If_Then_Else. Both X1 and X2 have the same constant value.

82.12.2 Overloading

In C, function names must be unique. Ada allows overloading, in which multiple subprograms can share the same name as long as the subprogram signatures (the parameter types, and function return types) are different. The compiler will be able to resolve the calls to the proper routines or it will reject the calls. For example:

[Ada]

Listing 2164: machine.ads

```
1 package Machine is
2     type Status is (Off, On);
3     type Code is new Integer range 0 .. 3;
4     type Threshold is new Float range 0.0 .. 10.0;
5
6     function Get (S : Status) return Code;
7     function Get (S : Status) return Threshold;
8
9 end Machine;
```

Listing 2165: machine.adb

```
1 package body Machine is
2
3     function Get (S : Status) return Code is
4     begin
5         case S is
6             when Off => return 1;
7             when On  => return 3;
8         end case;
9     end Get;
```

(continues on next page)

(continued from previous page)

```

10
11     function Get (S : Status) return Threshold is
12     begin
13         case S is
14             when Off => return 2.0;
15             when On  => return 10.0;
16         end case;
17     end Get;
18
19 end Machine;

```

Listing 2166: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Machine;      use Machine;
3
4  procedure Main is
5      S : Status;
6      C : Code;
7      T : Threshold;
8  begin
9      S := On;
10     C := Get (S);
11     T := Get (S);
12
13     Put_Line ("S: " & Status'Image (S));
14     Put_Line ("C: " & Code'Image (C));
15     Put_Line ("T: " & Threshold'Image (T));
16 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overloading_Ada
MD5: 909cdf00b629917f7131489702cc26f1

Runtime output

```

S: ON
C:  3
T:  1.00000E+01

```

The Ada compiler knows that an assignment to C requires a Code value. So, it chooses the Get function that returns a Code to satisfy this requirement.

Operators in Ada are functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To declare an operator as a function, enclose its "name" in quotes:

[Ada]

Listing 2167: machine_2.ads

```

1  package Machine_2 is
2      type Status is (Off, Waiting, On);
3      type Input is new Float range 0.0 .. 10.0;
4
5      function Get (I : Input) return Status;
6
7      function "=" (Left : Input; Right : Status) return Boolean;
8
9  end Machine_2;

```

Listing 2168: machine_2.adb

```

1 package body Machine_2 is
2
3     function Get (I : Input) return Status is
4     begin
5         if I >= 0.0 and I < 3.0 then
6             return Off;
7         elsif I >= 3.0 and I < 6.5 then
8             return Waiting;
9         else
10            return On;
11        end if;
12    end Get;
13
14    function "=" (Left : Input; Right : Status) return Boolean is
15    begin
16        return Get (Left) = Right;
17    end "=";
18
19 end Machine_2;

```

Listing 2169: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Machine_2;   use Machine_2;
3
4 procedure Main is
5     I : Input;
6 begin
7     I := 3.0;
8     if I = Off then
9         Put_Line ("Machine is off.");
10    else
11        Put_Line ("Machine is not off.");
12    end if;
13 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Overloading_Eq
MD5: c5580f15c1b93f73fff3afc147cd15a1

Runtime output

Machine is not off.

82.12.3 Aspects

Aspect specifications allow you to define certain characteristics of a declaration using the **with** keyword after the declaration:

```

procedure Some_Procedure is <procedure_definition>
    with Some_Aspect => <aspect_specification>;

function Some_Function is <function_definition>
    with Some_Aspect => <aspect_specification>;

type Some_Type is <type_definition>
    with Some_Aspect => <aspect_specification>;

```

(continues on next page)

(continued from previous page)

```
Obj : Some_Type with Some_Aspect => <aspect_specification>;
```

For example, you can inline a subprogram by specifying the Inline aspect:

[Ada]

Listing 2170: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is array (Positive range <>) of Float;
4
5     function Average (Data : Float_Array) return Float
6         with Inline;
7
8 end Float_Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Inline_Aspect
MD5: 6e25e81e4015d907d50aa9cf4a0a3fab

We'll discuss inlining *later in this course* (page 1944).

Aspect specifications were introduced in Ada 2012. In previous versions of Ada, you had to use a **pragma** instead. The previous example would be written as follows:

[Ada]

Listing 2171: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is array (Positive range <>) of Float;
4
5     function Average (Data : Float_Array) return Float;
6
7     pragma Inline (Average);
8
9 end Float_Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Inline_Aspect
MD5: bd5df14dce9577a054f0ec612d5bbe40

Aspects and attributes might refer to the same kind of information. For example, we can use the Size aspect to define the expected minimum size of objects of a certain type:

[Ada]

Listing 2172: my_device_types.ads

```
1 package My_Device_Types is
2
3     type UInt10 is mod 2 ** 10
4         with Size => 10;
5
6 end My_Device_Types;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Size_Aspect
MD5: 049be992b876dba42cf091afc256db35

In the same way, we can use the size attribute to retrieve the size of a type or of an object:
[Ada]

Listing 2173: show_device_types.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with My_Device_Types; use My_Device_Types;
4
5 procedure Show_Device_Types is
6   UInt10_Obj : constant UInt10 := 0;
7 begin
8   Put_Line ("Size of UInt10 type:  " & Positive'Image (UInt10'Size));
9   Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
10 end Show_Device_Types;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Size_Aspect
MD5: 4e46ad9cf54276b381b960672daa03b9

Runtime output

```
Size of UInt10 type:    10
Size of UInt10 object:  16
```

We'll explain both Size aspect and Size attribute *later in this course* (page 1835).

CONCURRENCY AND REAL-TIME

83.1 Understanding the various options

Concurrent and real-time programming are standard parts of the Ada language. As such, they have the same semantics, whether executing on a native target with an OS such as Linux, on a real-time operating system (RTOS) such as VxWorks, or on a bare metal target with no OS or RTOS at all.

For resource-constrained systems, two subsets of the Ada concurrency facilities are defined, known as the Ravenscar and Jorvik profiles. Though restricted, these subsets have highly desirable properties, including: efficiency, predictability, analyzability, absence of dead-lock, bounded blocking, absence of priority inversion, a real-time scheduler, and a small memory footprint. On bare metal systems, this means in effect that Ada comes with its own real-time kernel.

For further information

We'll discuss the Ravenscar profile *later in this chapter* (page 1817). Details about the Jorvik profile can be found elsewhere [[Jorvik](#)].

Enhanced portability and expressive power are the primary advantages of using the standard concurrency facilities, potentially resulting in considerable cost savings. For example, with little effort, it is possible to migrate from Windows to Linux to a bare machine without requiring any changes to the code. Thread management and synchronization is all done by the implementation, transparently. However, in some situations, it's critical to be able to access directly the services provided by the platform. In this case, it's always possible to make direct system calls from Ada code. Several targets of the GNAT compiler provide this sort of API by default, for example win32ada for Windows and Florist for POSIX systems.

On native and RTOS-based platforms GNAT typically provides the full concurrency facilities. In contrast, on bare metal platforms GNAT typically provides the two standard subsets: Ravenscar and Jorvik.

83.2 Tasks

Ada offers a high level construct called a *task* which is an independent thread of execution. In GNAT, tasks are either mapped to the underlying OS threads, or use a dedicated kernel when not available.

The following example will display the 26 letters of the alphabet twice, using two concurrent tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

Listing 2174: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is -- implicitly called by the environment task
4     subtype A_To_Z is Character range 'A' .. 'Z';
5
6     task My_Task;
7
8     task body My_Task is
9     begin
10         for I in A_To_Z'Range loop
11             Put (I);
12         end loop;
13         New_Line;
14     end My_Task;
15 begin
16     for I in A_To_Z'Range loop
17         Put (I);
18     end loop;
19     New_Line;
20 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.My_Task
MD5: 154702197f0c02f5750838e51a99f548

Runtime output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the begin. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

Task types can be parameterized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the underlying implementation of the task scheduling algorithm.

[Ada]

Listing 2175: my_tasks.ads

```
1 package My_Tasks is
2
3     task type My_Task (First : Character);
4
5 end My_Tasks;
```

Listing 2176: my_tasks.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Tasks is
4
5     task body My_Task is
6     begin
7         for I in First .. 'Z' loop
8             Put (I);
9         end loop;
10        New_Line;
11    end My_Task;
12
13 end My_Tasks;

```

Listing 2177: main.adb

```

1 with My_Tasks; use My_Tasks;
2
3 procedure Main is
4     Dummy_Tab : array (0 .. 3) of My_Task ('W');
5 begin
6     null;
7 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.My_Task_Type
MD5: 81d88397b0548fdcc1ba31549a8de4fd

Runtime output

```

WXYZ
WXYZ
WXYZ
WXYZ

```

In Ada, a task may be dynamically allocated rather than declared statically. The task will then start as soon as it has been allocated, and terminates when its work is completed.

[Ada]

Listing 2178: main.adb

```

1 with My_Tasks; use My_Tasks;
2
3 procedure Main is
4     type Ptr_Task is access My_Task;
5
6     T : Ptr_Task;
7 begin
8     T := new My_Task ('W');
9 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.My_Task_Type
MD5: d88a96eecf50ebbcdf9cb870f232a09

Runtime output

WXYZ

83.3 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Let's consider the following example:

[Ada]

Listing 2179: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5     task After is
6         entry Go;
7     end After;
8
9     task body After is
10        begin
11            accept Go;
12            Put_Line ("After");
13        end After;
14
15    begin
16        Put_Line ("Before");
17        After.Go;
18    end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Rendezvous
MD5: b0a595b1eecac793e40b6d1d41171766

Runtime output

Before
After

The Go entry declared in After is the client interface to the task. In the task body, the **accept** statement causes the task to wait for a call on the entry. This particular **entry** and **accept** pair simply causes the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The **entry/accept** pair can take/pass parameters, and the **accept** statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

[Ada]

Listing 2180: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
```

(continues on next page)

(continued from previous page)

```

5  task After is
6      entry Go (Text : String);
7  end After;
8
9  task body After is
10 begin
11     accept Go (Text : String) do
12         Put_Line ("After: " & Text);
13     end Go;
14 end After;
15
16 begin
17     Put_Line ("Before");
18     After.Go ("Main");
19 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Rendezvous_Params
MD5: 6430e88f5ae349128bb1f1d53f36251e

Runtime output

```

Before
After: Main

```

In the above example, the `Put_Line` is placed in the `accept` statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the begin of `Main`, task `After` is started and the main procedure is suspended.
2. `After` reaches the **accept** statement and is suspended, since there is no pending call on the `Go` entry.
3. The main procedure is awakened and executes the `Put_Line` invocation, displaying the string `"Before"`.
4. The main procedure calls the `Go` entry. Since `After` is suspended on its **accept** statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task `After` is awakened to execute the body of the **accept** statement. The actual parameter `"Main"` is passed to the **accept** statement, and the `Put_Line` invocation is executed. As a result, the string `"After: Main"` is displayed.
6. When the **accept** statement is completed, both the `After` task and the main procedure are ready to run. Suppose that the `Main` procedure is given the processor. It reaches its end, but the local task `After` has not yet terminated. The main procedure is suspended.
7. The `After` task continues, and terminates since it is at its end. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

83.4 Selective Rendezvous

The **accept** statement by itself can only wait for a single event (call) at a time. The **select** statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call `Increment`, `Decrement`, and `Get`:

[Ada]

Listing 2181: counters.ads

```
1 package Counters is
2
3     task Counter is
4         entry Get (Result : out Integer);
5         entry Increment;
6         entry Decrement;
7     end Counter;
8
9 end Counters;
```

Listing 2182: counters.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Counters is
4
5     task body Counter is
6         Value : Integer := 0;
7     begin
8         loop
9             select
10                 accept Increment do
11                     Value := Value + 1;
12                 end Increment;
13             or
14                 accept Decrement do
15                     Value := Value - 1;
16                 end Decrement;
17             or
18                 accept Get (Result : out Integer) do
19                     Result := Value;
20                 end Get;
21             or
22                 delay 5.0;
23                 Put_Line ("Exiting Counter task...");
24                 exit;
25             end select;
26         end loop;
27     end Counter;
28
29 end Counters;
```

Listing 2183: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Counters; use Counters;
3
4 procedure Main is
5     V : Integer;
6 begin
7     Put_Line ("Main started.");
8
9     Counter.Get (V);
10    Put_Line ("Got value. Value = " & Integer'Image (V));
11
12    Counter.Increment;
13    Put_Line ("Incremented value.");
14
```

(continues on next page)

(continued from previous page)

```

15   Counter.Increment;
16   Put_Line ("Incremented value.");
17
18   Counter.Get (V);
19   Put_Line ("Got value. Value = " & Integer'Image (V));
20
21   Counter.Decrement;
22   Put_Line ("Decrement value.");
23
24   Counter.Get (V);
25   Put_Line ("Got value. Value = " & Integer'Image (V));
26
27   Put_Line ("Main finished.");
28 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Selective_Rendezvous
MD5: 619d009bcfcd8053bc132b2e32a29249

Runtime output

```

Main started.
Got value. Value = 0
Incremented value.
Incremented value.
Got value. Value = 2
Decrement value.
Got value. Value = 1
Main finished.
Exiting Counter task...

```

When the task's statement flow reaches the `select`, it will wait for all four events — three entries and a delay — in parallel. If the delay of five seconds is exceeded, the task will execute the statements following the `delay` statement (and in this case will exit the loop, in effect terminating the task). The `accept` bodies for the Increment, Decrement, or Get entries will be otherwise executed as they're called. These four sections of the `select` statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a *race condition* could occur where multiple tasks simultaneously calling, say, Increment, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call Increment then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended at the `select` statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the Counter task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the five second delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the `select` branch with an `accept` statement for that entry.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued. The choice of which caller to accept depends on the queuing policy, which can be specified

via a **pragma** defined in the Real-Time Systems Annex of the Ada standard; the default is *First-In First-Out*.

83.5 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a protected object, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its protected operations) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

[Ada]

Listing 2184: counters.ads

```
1 package Counters is
2
3     protected Counter is
4         function Get return Integer;
5         procedure Increment;
6         procedure Decrement;
7     private
8         Value : Integer := 0;
9     end Counter;
10
11 end Counters;
```

Listing 2185: counters.adb

```
1 package body Counters is
2
3     protected body Counter is
4         function Get return Integer is
5             begin
6                 return Value;
7             end Get;
8
9         procedure Increment is
10             begin
11                 Value := Value + 1;
12             end Increment;
13
14         procedure Decrement is
15             begin
16                 Value := Value - 1;
17             end Decrement;
18     end Counter;
19
20 end Counters;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Protected_Counter
 MD5: f29f21621dfcf092580f6a130101788e

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

[Ada]

Listing 2186: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Counters;    use Counters;
3
4 procedure Main is
5 begin
6   Counter.Increment;
7   Counter.Decrement;
8   Put_Line (Integer'Image (Counter.Get));
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Protected_Counter
 MD5: 704e3a382fe38caa11ecd3d46fcd2beb

Runtime output

0

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```

declare
  protected type Counter is
    -- as above
  end Counter;

  protected body Counter is
    -- as above
  end Counter;

  C1 : Counter;
  C2 : Counter;
begin
  C1.Increment;
  C2.Decrement;
  . . .
end;
```

Protected objects and types can declare a procedure-like operation known as an *entry*. An entry is somewhat similar to a procedure but includes a so-called barrier condition that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition

is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

[Ada]

Listing 2187: binary_semaphores.ads

```
1 package Binary_Semaphores is
2
3     protected type Binary_Semaphore is
4         entry Wait;
5         procedure Signal;
6     private
7         Signaled : Boolean := False;
8     end Binary_Semaphore;
9
10 end Binary_Semaphores;
```

Listing 2188: binary_semaphores.adb

```
1 package body Binary_Semaphores is
2
3     protected body Binary_Semaphore is
4         entry Wait when Signaled is
5             begin
6                 Signaled := False;
7             end Wait;
8
9         procedure Signal is
10            begin
11                Signaled := True;
12            end Signal;
13        end Binary_Semaphore;
14
15    end Binary_Semaphores;
```

Listing 2189: main.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Binary_Semaphores; use Binary_Semaphores;
3
4 procedure Main is
5     B : Binary_Semaphore;
6
7     task T1;
8     task T2;
9
10    task body T1 is
11        begin
12            Put_Line ("Task T1 waiting...");
13            B.Wait;
14
15            Put_Line ("Task T1.");
16            delay 1.0;
17
18            Put_Line ("Task T1 will signal...");
19            B.Signal;
20
```

(continues on next page)

(continued from previous page)

```

21     Put_Line ("Task T1 finished.");
22 end T1;
23
24 task body T2 is
25 begin
26     Put_Line ("Task T2 waiting...");
27     B.Wait;
28
29     Put_Line ("Task T2");
30     delay 1.0;
31
32     Put_Line ("Task T2 will signal...");
33     B.Signal;
34
35     Put_Line ("Task T2 finished.");
36 end T2;
37
38 begin
39     Put_Line ("Main started.");
40     B.Signal;
41     Put_Line ("Main finished.");
42 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Protected_Binary_Semaphore
MD5: aa064a9ec056d44c4217e64cd05726a4

Runtime output

```

Task T2 waiting...
Main started.
Main finished.
Task T2
Task T1 waiting...
Task T2 will signal...
Task T2 finished.
Task T1.
Task T1 will signal...
Task T1 finished.

```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

83.6 Ravenscar

The Ravenscar profile is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are intended:

- hard real-time applications requiring predictability,
- safety-critical systems requiring formal, stringent certification,
- high-integrity applications requiring formal static analysis and verification,
- embedded applications requiring both a small memory footprint and low execution overhead.

Tasking constructs that preclude analysis, either technically or economically, are disallowed. You can use the **pragma Profile** (Ravenscar) to indicate that the Ravenscar

restrictions must be observed in your program.

Some of the examples we've seen above will be rejected by the compiler when using the Ravenscar profile. For example:

[Ada]

Listing 2190: my_tasks.ads

```
1 package My_Tasks is
2
3     task type My_Task (First : Character);
4
5 end My_Tasks;
```

Listing 2191: my_tasks.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Tasks is
4
5     task body My_Task is
6     begin
7         for C in First .. 'Z' loop
8             Put (C);
9         end loop;
10        New_Line;
11    end My_Task;
12
13 end My_Tasks;
```

Listing 2192: main.adb

```
1 pragma Profile (Ravenscar);
2
3 with My_Tasks; use My_Tasks;
4
5 procedure Main is
6     Tab : array (0 .. 3) of My_Task ('W');
7 begin
8     null;
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Ravenscar
MD5: b7518a039c2b4cecece1de63eeaa208f

Build output

```
main.adb:6:04: error: violation of restriction "No_Task_Hierarchy"
main.adb:6:04: error: from profile "Ravenscar" at line 1
gprbuild: *** compilation phase failed
```

This code violates the *No_Task_Hierarchy* restriction of the Ravenscar profile. This is due to the declaration of `Tab` in the `Main` procedure. Ravenscar requires task declarations to be done at the library level. Therefore, a simple solution is to create a separate package and reference it in the main application:

[Ada]

Listing 2193: my_task_inst.ads

```

1 with My_Tasks; use My_Tasks;
2
3 package My_Task_Inst is
4
5     Tab : array (0 .. 3) of My_Task ('W');
6
7 end My_Task_Inst;
```

Listing 2194: main.adb

```

1 pragma Profile (Ravenscar);
2
3 with My_Task_Inst;
4
5 procedure Main is
6 begin
7     null;
8 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Concurrency.Ravenscar
MD5: b38943dc1c962b5e691f2b6d9933a3ec

Runtime output

```

WXYZ
WXYZ
WXYZ
WXYZ
```

Also, Ravenscar prohibits entries for tasks. For example, we're not allowed to write this declaration:

```

task type My_Task (First : Character) is
    entry Start;
end My_Task;
```

You can use, however, one entry per protected object. As an example, the declaration of the Binary_Semaphore type that we've discussed before compiles fine with Ravenscar:

```

protected type Binary_Semaphore is
    entry Wait;
    procedure Signal;
private
    Signaled : Boolean := False;
end Binary_Semaphore;
```

We could add more procedures and functions to the declaration of Binary_Semaphore, but we wouldn't be able to add another entry when using Ravenscar.

Similar to the previous example with the task array declaration, objects of Binary_Semaphore cannot be declared in the main application:

```

procedure Main is
    B : Binary_Semaphore;
begin
    null;
end Main;
```

This violates the *No_Local_Protected_Objects* restriction. Again, Ravenscar expects this declaration to be done on a library level, so a solution to make this code compile is to have this declaration in a separate package and reference it in the Main procedure.

Ravenscar offers many additional restrictions. Covering those would exceed the scope of this chapter. You can find more examples using the Ravenscar profile on [this blog post](#)⁴³⁹.

⁴³⁹ <https://blog.adacore.com/theres-a-mini-rtos-in-my-language>

WRITING ADA ON EMBEDDED SYSTEMS

84.1 Understanding the Ada Run-Time

Ada supports a high level of abstractness and expressiveness. In some cases, the compiler translates those constructs directly into machine code. However, there are many high-level constructs for which a direct compilation would be difficult. In those cases, the compiler links to a library containing an implementation of those high-level constructs: this is the so-called run-time library.

One typical example of high-level constructs that can be cumbersome for direct machine code generation is Ada source-code using tasking. In this case, linking to a low-level implementation of multithreading support — for example, an implementation using POSIX threads — is more straightforward than trying to make the compiler generate all the machine code.

In the case of GNAT, the run-time library is implemented using both C and Ada source-code. Also, depending on the operating system, the library will interface with low-level functionality from the target operating system.

There are basically two types of run-time libraries:

- the **standard** run-time library: in many cases, this is the run-time library available on desktop operating systems or on some embedded platforms (such as ARM-Linux on a Raspberry-Pi).
- the **configurable** run-time library: this is a capability that is used to create custom run-time libraries for specific target devices.

Configurable run-time libraries are usually used for constrained target devices where support for the full library would be difficult or even impossible. In this case, configurable run-time libraries may support just a subset of the full Ada language. There are many reasons that speak for this approach:

- Some aspects of the Ada language may not translate well to limited operating systems.
- Memory constraints may require reducing the size of the run-time library, so that developers may need to replace or even remove parts of the library.
- When certification is required, those parts of the library that would require too much certification effort can be removed.

When using a configurable run-time library, the compiler checks whether the library supports certain features of the language. If a feature isn't supported, the compiler will give an error message.

You can find further information about the run-time library on [this chapter of the GNAT User's Guide Supplement for Cross Platforms](#)⁴⁴⁰

⁴⁴⁰ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/the_gnat_configurable_run_time_facility.html

84.2 Low Level Programming

84.2.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
  V  : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record
with Pack;
```

[C]

```
struct R {
    unsigned int v:8;
    bool b1;
    bool b2;
};
```

The Ada and the C code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C, the number of bits required by each field needs to be specified. Here, we're stating that *v* is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The *Pack* aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components. We'll see more details about the *Pack* aspect in the sections about *bitwise operations* (page 1875) and *mapping structures to bit-fields* (page 1878) in chapter 6.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

[Ada]

```
type R is record
  V  : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record;

for R use record
  -- Occupy the first bit of the first byte.
  B1 at 0 range 0 .. 0;
```

(continues on next page)

(continued from previous page)

```

-- Occupy the last 7 bits of the first byte,
-- as well as the first bit of the second byte.
V at 0 range 1 .. 8;

-- Occupy the second bit of the second byte.
B2 at 1 range 1 .. 1;
end record;

```

We omit the **with** Pack directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type R across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

84.2.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the `rdtsc` instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64-bit value split between the `edx` and `eax` registers.

GNAT provides a subprogram called `System.Machine_Code.Asm` that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

[Ada]

Listing 2195: `get_processor_cycles.adb`

```

1 with System.Machine_Code; use System.Machine_Code;
2 with Interfaces;         use Interfaces;
3
4 function Get_Processor_Cycles return Unsigned_64 is
5   Low, High : Unsigned_32;
6   Counter   : Unsigned_64;
7 begin
8   Asm ("rdtsc",
9       Outputs =>
10         (Unsigned_32'Asm_Output ("=a", High),
11          Unsigned_32'Asm_Output ("=d", Low)),
12       Volatile => True);
13
14   Counter :=
15     Unsigned_64 (High) * 2 ** 32 +
16     Unsigned_64 (Low);
17
18   return Counter;
19 end Get_Processor_Cycles;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Assembly_Code
 MD5: 092be19e223946ebb9fb9f4786003b94

The `Unsigned_32'Asm_Output` clauses above provide associations between machine registers and source-level variables to be updated. `=a` and `=d` refer to the `eax` and `edx` machine registers, respectively. The use of the `Unsigned_32` and `Unsigned_64` types from package `Interfaces` ensures correct representation of the data. We assemble the two 32-bit values to form a single 64-bit value.

We set the `Volatile` parameter to `True` to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the `eax` and `edx` registers to implement the `High` and `Low` variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

84.3 Interrupt Handling

Handling interrupts is an important aspect when programming embedded devices. Interrupts are used, for example, to indicate that a hardware or software event has happened. Therefore, by handling interrupts, an application can react to external events.

Ada provides built-in support for handling interrupts. We can process interrupts by attaching a handler — which must be a protected procedure — to it. In the declaration of the protected procedure, we use the `Attach_Handler` aspect and indicate which interrupt we want to handle.

Let's look into a code example that *traps* the quit interrupt (`SIGQUIT`) on Linux:

[Ada]

Listing 2196: `signal_handlers.ads`

```
1 with System.OS_Interface;
2
3 package Signal_Handlers is
4
5     protected type Quit_Handler is
6         function Requested return Boolean;
7     private
8         Quit_Request : Boolean := False;
9
10        --
11        -- Declaration of an interrupt handler for the "quit" interrupt:
12        --
13        procedure Handle_Quit_Signal
14            with Attach_Handler => System.OS_Interface.SIGQUIT;
15    end Quit_Handler;
16
17 end Signal_Handlers;
```

Listing 2197: `signal_handlers.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Signal_Handlers is
4
5     protected body Quit_Handler is
6
7         function Requested return Boolean is
```

(continues on next page)

(continued from previous page)

```

8      (Quit_Request);
9
10     procedure Handle_Quit_Signal is
11     begin
12         Put_Line ("Quit request detected!");
13         Quit_Request := True;
14     end Handle_Quit_Signal;
15
16 end Quit_Handler;
17
18 end Signal_Handlers;

```

Listing 2198: test_quit_handler.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Signal_Handlers;
3
4  procedure Test_Quit_Handler is
5      Quit : Signal_Handlers.Quit_Handler;
6
7  begin
8      while True loop
9          delay 1.0;
10         exit when Quit.Requested;
11     end loop;
12
13     Put_Line ("Exiting application...");
14 end Test_Quit_Handler;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Quit_Handler
MD5: d272c5bc59576444e09007a04a615ccf

The specification of the `Signal_Handlers` package from this example contains the declaration of `Quit_Handler`, which is a protected type. In the private part of this protected type, we declare the `Handle_Quit_Signal` procedure. By using the `Attach_Handler` aspect in the declaration of `Handle_Quit_Signal` and indicating the quit interrupt (`System.OS_Interface.SIGQUIT`), we're instructing the operating system to call this procedure for any quit request. So when the user presses `CTRL+\` on their keyboard, for example, the application will behave as follows:

- the operating system calls the `Handle_Quit_Signal` procedure, which displays a message to the user ("Quit request detected!") and sets a Boolean variable — `Quit_Request`, which is declared in the `Quit_Handler` type;
- the main application checks the status of the quit handler by calling the `Requested` function as part of the `while True` loop;
 - This call is in the `exit when Quit.Requested` line.
 - The `Requested` function returns `True` in this case because the `Quit_Request` flag was set by the `Handle_Quit_Signal` procedure.
- the main applications exits the loop, displays a message and finishes.

Note that the code example above isn't portable because it makes use of interrupts from the Linux operating system. When programming embedded devices, we would use instead the interrupts available on those specific devices.

Also note that, in the example above, we're declaring a static handler at compilation time. If you need to make use of dynamic handlers, which can be configured at runtime, you can

use the subprograms from the `Ada.Interrupts` package. This package includes not only a version of `Attach_Handler` as a procedure, but also other procedures such as:

- `Exchange_Handler`, which lets us exchange, at runtime, the current handler associated with a specific interrupt by a different handler;
- `Detach_Handler`, which we can use to remove the handler currently associated with a given interrupt.

Details about the `Ada.Interrupts` package are out of scope for this course. We'll discuss them in a separate, more advanced course in the future. You can find some information about it in the [Interrupts appendix of the Ada Reference Manual](#)⁴⁴¹.

84.4 Dealing with Absence of FPU with Fixed Point

Many numerical applications typically use floating-point types to compute values. However, in some platforms, a floating-point unit may not be available. Other platforms may have a floating-point unit, but using it in certain numerical algorithms can be prohibitive in terms of performance. For those cases, fixed-point arithmetic can be a good alternative.

The difference between fixed-point and floating-point types might not be so obvious when looking at this code snippet:

[Ada]

Listing 2199: `fixed_definitions.ads`

```
1 package Fixed_Definitions is
2
3   D : constant := 2.0 ** (-31);
4
5   type Fixed is delta D range -1.0 .. 1.0 - D;
6
7 end Fixed_Definitions;
```

Listing 2200: `show_float_and_fixed_point.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Fixed_Definitions; use Fixed_Definitions;
4
5 procedure Show_Float_And_Fixed_Point is
6   Float_Value : Float := 0.25;
7   Fixed_Value : Fixed := 0.25;
8 begin
9
10  Float_Value := Float_Value + 0.25;
11  Fixed_Value := Fixed_Value + 0.25;
12
13  Put_Line ("Float_Value = " & Float'Image (Float_Value));
14  Put_Line ("Fixed_Value = " & Fixed'Image (Fixed_Value));
15 end Show_Float_And_Fixed_Point;
```

Code block metadata

Project: `Courses.Ada_For_Embedded_C_Dev.Embedded.Fixed_Point`
MD5: 881817bb310304bc285f01454ab446f7

Runtime output

⁴⁴¹ <http://www.ada-auth.org/standards/12aarm/html/AA-C-3-2.html>

```
Float_Value = 5.00000E-01
Fixed_Value = 0.5000000000
```

In this example, the application will show the value 0.5 for both `Float_Value` and `Fixed_Value`.

The major difference between floating-point and fixed-point types is in the way the values are stored. Values of ordinary fixed-point types are, in effect, scaled integers. The scaling used for ordinary fixed-point types is defined by the type's `small`, which is derived from the specified `delta` and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types. In that sense, ordinary fixed-point types can be thought of being close to the actual representation on the machine. In fact, ordinary fixed-point types make use of the available integer shift instructions, for example.

Another difference between floating-point and fixed-point types is that Ada doesn't provide standard fixed-point types — except for the `Duration` type, which is used to represent an interval of time in seconds. While the Ada standard specifies floating-point types such as `Float` and `Long_Float`, we have to declare our own fixed-point types. Note that, in the previous example, we have used a fixed-point type named `Fixed`: this type isn't part of the standard, but must be declared somewhere in the source-code of our application.

The syntax for an ordinary fixed-point type is

```
type <type_name> is delta <delta_value> range <lower_bound> .. <upper_bound>;
```

By default, the compiler will choose a scale factor, or `small`, that is a power of 2 no greater than `<delta_value>`.

For example, we may define a normalized range between -1.0 and 1.0 as following:

[Ada]

Listing 2201: `normalized_fixed_point_type.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Normalized_Fixed_Point_Type is
4   D : constant := 2.0 ** (-31);
5   type TQ31 is delta D range -1.0 .. 1.0 - D;
6 begin
7   Put_Line ("TQ31 requires " & Integer'Image (TQ31'Size) & " bits");
8   Put_Line ("The delta value of TQ31 is " & TQ31'Image (TQ31'Delta));
9   Put_Line ("The minimum value of TQ31 is " & TQ31'Image (TQ31'First));
10  Put_Line ("The maximum value of TQ31 is " & TQ31'Image (TQ31'Last));
11 end Normalized_Fixed_Point_Type;
```

Code block metadata

Project: `Courses.Ada_For_Embedded_C_Dev.Embedded.Normalized_Fixed_Point_Type`
MD5: `2fe6e9f9bd20d2cfab959d1c0273280b`

Runtime output

```
TQ31 requires 32 bits
The delta value of TQ31 is 0.0000000005
The minimum value of TQ31 is -1.0000000000
The maximum value of TQ31 is 0.9999999995
```

In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not exactly one. This is a typical effect of fixed-point data types — you can find more details

in this discussion about the [Q format](#)⁴⁴². We may also rewrite this code with an exact type definition:

[Ada]

Listing 2202: normalized_adapted_fixed_point_type.ads

```
1 package Normalized_Adapted_Fixed_Point_Type is
2
3     type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
4
5 end Normalized_Adapted_Fixed_Point_Type;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Normalized_Adapted_Fixed_Point_Type
MD5: abe5f4e029c7c3c7a069890882b17f50

We may also use any other range. For example:

[Ada]

Listing 2203: custom_fixed_point_range.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Numerics; use Ada.Numerics;
3
4 procedure Custom_Fixed_Point_Range is
5     type Inv_Trig is delta 2.0 ** (-15) * Pi range -Pi / 2.0 .. Pi / 2.0;
6 begin
7     Put_Line ("Inv_Trig requires " & Integer'Image (Inv_Trig'Size)
8               & " bits");
9     Put_Line ("The delta value of Inv_Trig is "
10              & Inv_Trig'Image (Inv_Trig'Delta));
11     Put_Line ("The minimum value of Inv_Trig is "
12              & Inv_Trig'Image (Inv_Trig'First));
13     Put_Line ("The maximum value of Inv_Trig is "
14              & Inv_Trig'Image (Inv_Trig'Last));
15 end Custom_Fixed_Point_Range;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Custom_Fixed_Point_Range
MD5: 0d9a4bc96191d1341bbb1c081555b613

Runtime output

```
Inv_Trig requires 16 bits
The delta value of Inv_Trig is 0.00006
The minimum value of Inv_Trig is -1.57080
The maximum value of Inv_Trig is 1.57080
```

In this example, we are defining a 16-bit type called `Inv_Trig`, which has a range from $-\pi/2$ to $\pi/2$.

All standard operations are available for fixed-point types. For example:

[Ada]

⁴⁴² [https://en.wikipedia.org/wiki/Q_\(number_format\)](https://en.wikipedia.org/wiki/Q_(number_format))

Listing 2204: fixed_point_op.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Fixed_Point_Op is
4   type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
5
6   A, B, R : TQ31;
7 begin
8   A := 0.25;
9   B := 0.50;
10  R := A + B;
11  Put_Line ("R is " & TQ31'Image (R));
12 end Fixed_Point_Op;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Fixed_Point_Op
MD5: 78bafd93b25da898c00cc38c9d518e2a

Runtime output

R is 0.7500000000

As expected, R contains 0.75 after the addition of A and B.

In the case of C, since the language doesn't support fixed-point arithmetic, we need to emulate it using integer types and custom operations via functions. Let's look at this very rudimentary example:

[C]

Listing 2205: main.c

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define SHIFT_FACTOR 32
5
6 #define TO_FIXED(x) ((int) ((x) * pow (2.0, SHIFT_FACTOR - 1)))
7 #define TO_FLOAT(x) ((float) ((double)(x) * (double)pow (2.0, -(SHIFT_FACTOR -
8 ↵1))))
9
10 typedef int fixed;
11
12 fixed add (fixed a, fixed b)
13 {
14   return a + b;
15 }
16
17 fixed mult (fixed a, fixed b)
18 {
19   return (fixed)(((long)a * (long)b) >> (SHIFT_FACTOR - 1));
20 }
21
22 void display_fixed (fixed x)
23 {
24   printf("value (integer) = %d\n", x);
25   printf("value (float) = %3.5f\n\n", TO_FLOAT (x));
26 }
27
28 int main(int argc, const char * argv[])

```

(continues on next page)

(continued from previous page)

```
28 {  
29   int fixed_value = T0_FIXED(0.25);  
30  
31   printf("Original value\n");  
32   display_fixed(fixed_value);  
33  
34   printf("... + 0.25\n");  
35   fixed_value = add(fixed_value, T0_FIXED(0.25));  
36   display_fixed(fixed_value);  
37  
38   printf("... * 0.5\n");  
39   fixed_value = mult(fixed_value, T0_FIXED(0.5));  
40   display_fixed(fixed_value);  
41  
42   return 0;  
43 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Fixed_Point_C
MD5: 61016e8fc0dbc4d0eefd2c86915489e5

Runtime output

```
Original value  
value (integer) = 536870912  
value (float)   = 0.25000  
  
... + 0.25  
value (integer) = 1073741824  
value (float)   = 0.50000  
  
... * 0.5  
value (integer) = 536870912  
value (float)   = 0.25000
```

Here, we declare the fixed-point type `fixed` based on `int` and two operations for it: addition (via the `add` function) and multiplication (via the `mult` function). Note that, while fixed-point addition is quite straightforward, multiplication requires right-shifting to match the correct internal representation. In Ada, since fixed-point operations are part of the language specification, they don't need to be emulated. Therefore, no extra effort is required from the programmer.

Also note that the example above is very rudimentary, so it doesn't take some of the side-effects of fixed-point arithmetic into account. In C, you have to manually take all side-effects deriving from fixed-point arithmetic into account, while in Ada, the compiler takes care of selecting the right operations for you.

84.5 Volatile and Atomic data

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

84.5.1 Volatile

A **volatile**⁴⁴³ object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread B may modify the value of that object between two read accesses of a thread A. Another typical example is the one of **memory-mapped I/O**⁴⁴⁴, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code that stores the value of that object into a register and use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example of a volatile variable in C:

[C]

Listing 2206: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      volatile double val = 0.0;
6      int i;
7
8      for (i = 0; i < 1000; i++)
9      {
10         val += i * 2.0;
11     }
12     printf ("val: %5.3f\n", val);
13
14     return 0;
15 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Object_C
MD5: 863c7dda4acb3286976a1edab29bab08

Runtime output

val: 999000.000

In this example, `val` has the modifier **volatile**, which indicates that the compiler must handle `val` as a volatile object. Therefore, each read and write access in the loop is performed by accessing the value of `val` in then memory.

This is the corresponding implementation in Ada:

[Ada]

Listing 2207: show_volatile_object.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Volatile_Object is
4      Val : Long_Float with Volatile;
```

(continues on next page)

⁴⁴³ [https://en.wikipedia.org/wiki/Volatile_\(computer_programming\)](https://en.wikipedia.org/wiki/Volatile_(computer_programming))

⁴⁴⁴ https://en.wikipedia.org/wiki/Memory-mapped_I/O

(continued from previous page)

```
5 begin
6   Val := 0.0;
7   for I in 0 .. 999 loop
8     Val := Val + 2.0 * Long_Float (I);
9   end loop;
10
11   Put_Line ("Val: " & Long_Float'Image (Val));
12 end Show_Volatile_Object;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Object_Ada
MD5: aa1e276e64e69813bfc3e3ef39f3dd47

Runtime output

Val: 9.990000000000000E+05

In this example, Val has the Volatile aspect, which makes the object volatile. We can also use the Volatile aspect in type declarations. For example:

[Ada]

Listing 2208: show_volatile_type.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Type is
4   type Volatile_Long_Float is new Long_Float with Volatile;
5
6   Val : Volatile_Long_Float;
7 begin
8   Val := 0.0;
9   for I in 0 .. 999 loop
10    Val := Val + 2.0 * Volatile_Long_Float (I);
11   end loop;
12
13   Put_Line ("Val: " & Volatile_Long_Float'Image (Val));
14 end Show_Volatile_Type;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Type
MD5: 41ecf028803a58ce244c421eaeb118e4

Runtime output

Val: 9.990000000000000E+05

Here, we're declaring a new type Volatile_Long_Float based on the Long_Float type and using the Volatile aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the Volatile_Components aspect in the array declaration. For example:

[Ada]

Listing 2209: show_volatile_array_components.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Volatile_Array_Components is
```

(continues on next page)

(continued from previous page)

```

4   Arr : array (1 .. 2) of Long_Float with Volatile_Components;
5   begin
6     Arr := (others => 0.0);
7
8     for I in 0 .. 999 loop
9       Arr (1) := Arr (1) + 2.0 * Long_Float (I);
10      Arr (2) := Arr (2) + 10.0 * Long_Float (I);
11    end loop;
12
13    Put_Line ("Arr (1): " & Long_Float'Image (Arr (1)));
14    Put_Line ("Arr (2): " & Long_Float'Image (Arr (2)));
15  end Show_Volatile_Array_Components;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Volatile_Array_Components
MD5: 601d61dd01888c60ae1a51ec513138d5

Runtime output

```

Arr (1):  9.990000000000000E+05
Arr (2):  4.995000000000000E+06

```

Note that it's possible to use the Volatile aspect for the array declaration as well:

[Ada]

```
Arr : array (1 .. 2) of Long_Float with Volatile;
```

84.5.2 Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

** In other contexts**

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store / load instructions, or even two 16-bit store / load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store / load instruction.

Because of this strict definition, we might have objects for which the Atomic aspect cannot be specified. Lots of machines support integer types that are larger than the native word-sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the `Atomic` aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:

[Ada]

Listing 2210: `show_shared_hw_register.adb`

```
1 with System;
2
3 procedure Show_Shared_HW_Register is
4   R : Integer
5     with Atomic, Address => System'To_Address (16#FFFF00A0#);
6 begin
7   null;
8 end Show_Shared_HW_Register;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Atomic_Object
MD5: 7ef148adf393819fc3fbc25eb45afe46

Note that the `Address` aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register. We'll discuss more about the `Address` aspect later in the section about *mapping structures to bit-fields* (page 1878) (in chapter 6).

In addition to atomic objects, we can declare atomic types and atomic array components — similarly to what we've seen before for volatile objects. For example:

[Ada]

Listing 2211: `show_shared_hw_register.adb`

```
1 with System;
2
3 procedure Show_Shared_HW_Register is
4   type Atomic_Integer is new Integer with Atomic;
5
6   R : Atomic_Integer with Address => System'To_Address (16#FFFF00A0#);
7
8   Arr : array (1 .. 2) of Integer with Atomic_Components;
9 begin
10  null;
11 end Show_Shared_HW_Register;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Atomic_Types_Arrays
MD5: 11475b5152087eff7f36abfe2c5ae9a1

In this example, we're declaring the `Atomic_Integer` type, which is an atomic type. Objects of this type — such as `R` in this example — are automatically atomic. This example also includes the declaration of the `Arr` array, which has atomic components.

84.6 Interfacing with Devices

Previously, we've seen that we can use *representation clauses* (page 1822) to specify a particular layout for a record type. As mentioned before, this is useful when interfacing with hardware, drivers, or communication protocols. In this section, we'll extend this concept

for two specific use-cases: register overlays and data streams. Before we discuss those use-cases, though, we'll first explain the Size aspect and the Size attribute.

84.6.1 Size aspect and attribute

The Size aspect indicates the minimum number of bits required to represent an object. When applied to a type, the Size aspect is telling the compiler to not make record or array components of a type T any smaller than X bits. Therefore, a common usage for this aspect is to just confirm expectations: developers specify 'Size to tell the compiler that T should fit X bits, and the compiler will tell them if they are right (or wrong).

When the specified size value is larger than necessary, it can cause objects to be bigger in memory than they would be otherwise. For example, for some enumeration types, we could say `for type Enum'Size use 32`; when the number of literals would otherwise have required only a byte. That's useful for unchecked conversions because the sizes of the two types need to be the same. Likewise, it's useful for interfacing with C, where `enum` types are just mapped to the `int` type, and thus larger than Ada might otherwise require. We'll discuss unchecked conversions *later in the course* (page 1891).

Let's look at an example from an earlier chapter:

[Ada]

Listing 2212: my_device_types.ads

```
1 package My_Device_Types is
2
3     type UInt10 is mod 2 ** 10
4       with Size => 10;
5
6 end My_Device_Types;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Size_Aspect
MD5: 049be992b876dba42cf091afc256db35

Here, we're saying that objects of type UInt10 must have at least 10 bits. In this case, if the code compiles, it is a confirmation that such values can be represented in 10 bits when packed into an enclosing record or array type.

If the size specified was larger than what the compiler would use by default, then it could affect the size of objects. For example, for UInt10, anything up to and including 16 would make no difference on a typical machine. However, anything over 16 would then push the compiler to use a larger object representation. That would be important for unchecked conversions, for example.

The Size attribute indicates the number of bits required to represent a type or an object. We can use the size attribute to retrieve the size of a type or of an object:

[Ada]

Listing 2213: show_device_types.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with My_Device_Types; use My_Device_Types;
4
5 procedure Show_Device_Types is
6   UInt10_Obj : constant UInt10 := 0;
7 begin
8   Put_Line ("Size of UInt10 type:  " & Positive'Image (UInt10'Size));
```

(continues on next page)

(continued from previous page)

```

9   Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
10  end Show_Device_Types;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Perspective.Size_Aspect
 MD5: 4e46ad9cf54276b381b960672daa03b9

Runtime output

```

Size of UInt10 type:    10
Size of UInt10 object: 16

```

Here, we're retrieving the actual sizes of the UInt10 type and an object of that type. Note that the sizes don't necessarily need to match. For example, although the size of UInt10 type is expected to be 10 bits, the size of UInt10_Obj may be 16 bits, depending on the platform. Also, components of this type within composite types (arrays, records) will probably be 16 bits as well unless they are packed.

84.6.2 Register overlays

Register overlays make use of representation clauses to create a structure that facilitates manipulating bits from registers. Let's look at a simplified example of a power management controller containing registers such as a system clock enable register. Note that this example is based on an actual architecture:

[Ada]

Listing 2214: registers.ads

```

1  with System;
2
3  package Registers is
4
5      type Bit    is mod 2 ** 1
6          with Size => 1;
7      type UInt5   is mod 2 ** 5
8          with Size => 5;
9      type UInt10  is mod 2 ** 10
10         with Size => 10;
11
12     subtype USB_Clock_Enable is Bit;
13
14     -- System Clock Enable Register
15     type PMC_SCER_Register is record
16         -- Reserved bits
17         Reserved_0_4    : UInt5           := 16#0#;
18         -- Write-only. Enable USB FS Clock
19         USBCLK          : USB_Clock_Enable := 16#0#;
20         -- Reserved bits
21         Reserved_6_15   : UInt10          := 16#0#;
22     end record
23     with
24         Volatile,
25         Size      => 16,
26         Bit_Order => System.Low_Order_First;
27
28     for PMC_SCER_Register use record
29         Reserved_0_4    at 0 range 0 .. 4;
30         USBCLK          at 0 range 5 .. 5;

```

(continues on next page)

(continued from previous page)

```

31     Reserved_6_15  at 0 range 6 .. 15;
32 end record;
33
34 -- Power Management Controller
35 type PMC_Peripheral is record
36     -- System Clock Enable Register
37     PMC_SCER       : aliased PMC_SCER_Register;
38     -- System Clock Disable Register
39     PMC_SCDR       : aliased PMC_SCER_Register;
40 end record
41     with Volatile;
42
43 for PMC_Peripheral use record
44     -- 16-bit register at byte 0
45     PMC_SCER       at 16#0# range 0 .. 15;
46     -- 16-bit register at byte 2
47     PMC_SCDR       at 16#2# range 0 .. 15;
48 end record;
49
50 -- Power Management Controller
51 PMC_Periph : aliased PMC_Peripheral
52     with Import, Address => System'To_Address (16#400E0600#);
53
54 end Registers;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.PMC_Peripheral
 MD5: d6f37976ca653d65d71ee5ea463df81c

First, we declare the system clock enable register — this is `PMC_SCER_Register` type in the code example. Most of the bits in that register are reserved. However, we're interested in bit #5, which is used to activate or deactivate the system clock. To achieve a correct representation of this bit, we do the following:

- We declare the `USBCLK` component of this record using the `USB_Clock_Enable` type, which has a size of one bit; and
- we use a representation clause to indicate that the `USBCLK` component is specifically at bit #5 of byte #0.

After declaring the system clock enable register and specifying its individual bits as components of a record type, we declare the power management controller type — `PMC_Peripheral` record type in the code example. Here, we declare two 16-bit registers as record components of `PMC_Peripheral`. These registers are used to enable or disable the system clock. The strategy we use in the declaration is similar to the one we've just seen above:

- We declare these registers as components of the `PMC_Peripheral` record type;
- we use a representation clause to specify that the `PMC_SCER` register is at byte #0 and the `PMC_SCDR` register is at byte #2.
 - Since these registers have 16 bits, we use a range of bits from 0 to 15.

The actual power management controller becomes accessible by the declaration of the `PMC_Periph` object of `PMC_Peripheral` type. Here, we specify the actual address of the memory-mapped registers (`400E0600` in hexadecimal) using the **Address** aspect in the declaration. When we use the **Address** aspect in an object declaration, we're indicating the address in memory of that object.

Because we specify the address of the memory-mapped registers in the declaration of `PMC_Periph`, this object is now an overlay for those registers. This also means that any

operation on this object corresponds to an actual operation on the registers of the power management controller. We'll discuss more details about overlays in the section about [mapping structures to bit-fields](#) (page 1878) (in chapter 6).

Finally, in a test application, we can access any bit of any register of the power management controller with simple record component selection. For example, we can set the USBCLK bit of the PMC_SCER register by using `PMC_Periph.PMC_SCER.USBCLK`:

[Ada]

Listing 2215: enable_usb_clock.adb

```
1 with Registers;
2
3 procedure Enable_USB_Clock is
4 begin
5     Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
6 end Enable_USB_Clock;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.PMC_Peripheral
MD5: b8f35a80d5f04cd362e5309aef33a100

This code example makes use of many aspects and keywords of the Ada language. One of them is the `Volatile` aspect, which we've discussed in the section about [volatile and atomic objects](#) (page 1830). Using the `Volatile` aspect for the `PMC_SCER_Register` type ensures that objects of this type won't be stored in a register.

In the declaration of the `PMC_SCER_Register` record type of the example, we use the `Bit_Order` aspect to specify the bit ordering of the record type. Here, we can select one of these options:

- `High_Order_First`: first bit of the record is the most significant bit;
- `Low_Order_First`: first bit of the record is the least significant bit.

The declarations from the `Registers` package also makes use of the `Import`, which is sometimes necessary when creating overlays. When used in the context of object declarations, it avoids default initialization (for data types that have it.). Aspect `Import` will be discussed in the section that explains how to [map structures to bit-fields](#) (page 1878) in chapter 6. Please refer to that chapter for more details.

Details about 'Size

In the example above, we're using the `Size` aspect in the declaration of the `PMC_SCER_Register` type. In this case, the effect is that it has the compiler confirm that the record type will fit into the expected 16 bits.

That's what the aspect does for type `PMC_SCER_Register` in the example above, as well as for the types `Bit`, `UInt5` and `UInt10`. For example, we may declare a stand-alone object of type `Bit`:

Listing 2216: show_bit_declaration.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Bit_Declaration is
4
5     type Bit    is mod 2 ** 1
6     with Size => 1;
7
8     B : constant Bit := 0;
9     -- ^ Although Bit'Size is 1, B'Size is almost certainly 8
10 begin
11     Put_Line ("Bit'Size = " & Positive'Image (Bit'Size));
12     Put_Line ("B'Size = " & Positive'Image (B'Size));
13 end Show_Bit_Declaration;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Bit_Declaration
 MD5: 1778bb96b4bf77292885bdeedfee7c596

Runtime output

```
Bit'Size = 1
B'Size   = 8
```

In this case, B is almost certainly going to be 8-bits wide on a typical machine, even though the language requires that Bit'Size is 1 by default.

In the declaration of the components of the PMC_Peripheral record type, we use the **aliased** keyword to specify that those record components are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This makes sense because we want to ensure that we're accessing specific memory-mapped registers, and not registers assigned by the compiler. Note that, for the same reason, we also use the **aliased** keyword in the declaration of the PMC_Periph object.

84.6.3 Data streams

Creating data streams — in the context of interfacing with devices — means the serialization of arbitrary information and its transmission over a communication channel. For example, we might want to transmit the content of memory-mapped registers as byte streams using a serial port. To do this, we first need to get a serialized representation of those registers as an array of bytes, which we can then transmit over the serial port.

Serialization of arbitrary record types — including register overlays — can be achieved by declaring an array of bytes as an overlay. By doing this, we're basically interpreting the information from those record types as bytes while ignoring their actual structure — i.e. their components and representation clause. We'll discuss details about overlays in the section about *mapping structures to bit-fields* (page 1878) (in chapter 6).

Let's look at a simple example of serialization of an arbitrary record type:

[Ada]

Listing 2217: arbitrary_types.ads

```
1 package Arbitrary_Types is
2
3     type Arbitrary_Record is record
4         A : Integer;
5         B : Integer;
6         C : Integer;
7     end record;
8
9 end Arbitrary_Types;
```

Listing 2218: serialize_data.ads

```
1 with Arbitrary_Types;
2
3 procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record);
```

Listing 2219: serialize_data.adb

```
1 with Arbitrary_Types;
2
3 procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record) is
    (continues on next page)
```

(continued from previous page)

```

4  type UByte is new Natural range 0 .. 255
5      with Size => 8;
6
7  type UByte_Array is array (Positive range <>) of UByte;
8
9  --
10 -- We can access the serialized data in Raw_TX, which is our overlay
11 --
12 Raw_TX : UByte_Array (1 .. Some_Object'Size / 8)
13     with Address => Some_Object'Address;
14 begin
15     null;
16     --
17     -- Now, we could stream the data from Some_Object.
18     --
19     -- For example, we could send the bytes (from Raw_TX) via the
20     -- serial port.
21     --
22 end Serialize_Data;

```

Listing 2220: data_stream_declaration.adb

```

1  with Arbitrary_Types;
2  with Serialize_Data;
3
4  procedure Data_Stream_Declaration is
5      Dummy_Object : Arbitrary_Types.Arbitrary_Record;
6
7  begin
8      Serialize_Data (Dummy_Object);
9  end Data_Stream_Declaration;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Data_Stream_Declaration
MD5: 1de6f518520010c28fd8deb29a2bf209

The most important part of this example is the implementation of the `Serialize_Data` procedure, where we declare `Raw_TX` as an overlay for our arbitrary object (`Some_Object` of `Arbitrary_Record` type). In simple terms, by writing `with Address => Some_Object'Address;` in the declaration of `Raw_TX`, we're specifying that `Raw_TX` and `Some_Object` have the same address in memory. Here, we are:

- taking the address of `Some_Object` — using the **Address** attribute —, and then
- using it as the address of `Raw_TX` — which is specified with the **Address** aspect.

By doing this, we're essentially saying that both `Raw_TX` and `Some_Object` are different representations of the same object in memory.

Because the `Raw_TX` overlay is completely agnostic about the actual structure of the record type, the `Arbitrary_Record` type could really be anything. By declaring `Raw_TX`, we create an array of bytes that we can use to stream the information from `Some_Object`.

We can use this approach and create a data stream for the register overlay example that we've seen before. This is the corresponding implementation:

[Ada]

Listing 2221: registers.ads

```

1  with System;
2
3  package Registers is
4
5      type Bit is mod 2 ** 1
6          with Size => 1;
7      type UInt5 is mod 2 ** 5
8          with Size => 5;
9      type UInt10 is mod 2 ** 10
10         with Size => 10;
11
12     subtype USB_Clock_Enable is Bit;
13
14     -- System Clock Register
15     type PMC_SCER_Register is record
16         -- Reserved bits
17         Reserved_0_4 : UInt5 := 16#0#;
18         -- Write-only. Enable USB FS Clock
19         USBCLK       : USB_Clock_Enable := 16#0#;
20         -- Reserved bits
21         Reserved_6_15 : UInt10 := 16#0#;
22     end record
23     with
24         Volatile,
25         Size      => 16,
26         Bit_Order => System.Low_Order_First;
27
28     for PMC_SCER_Register use record
29         Reserved_0_4 at 0 range 0 .. 4;
30         USBCLK       at 0 range 5 .. 5;
31         Reserved_6_15 at 0 range 6 .. 15;
32     end record;
33
34     -- Power Management Controller
35     type PMC_Peripheral is record
36         -- System Clock Enable Register
37         PMC_SCER : aliased PMC_SCER_Register;
38         -- System Clock Disable Register
39         PMC_SCDR : aliased PMC_SCER_Register;
40     end record
41     with Volatile;
42
43     for PMC_Peripheral use record
44         -- 16-bit register at byte 0
45         PMC_SCER at 16#0# range 0 .. 15;
46         -- 16-bit register at byte 2
47         PMC_SCDR at 16#2# range 0 .. 15;
48     end record;
49
50     -- Power Management Controller
51     PMC_Periph : aliased PMC_Peripheral;
52     -- with Import, Address => System'To_Address (16#400E0600#);
53
54 end Registers;

```

Listing 2222: serial_ports.ads

```

1  package Serial_Ports is
2

```

(continues on next page)

(continued from previous page)

```

3  type UByte is new Natural range 0 .. 255
4      with Size => 8;
5
6  type UByte_Array is array (Positive range <>) of UByte;
7
8  type Serial_Port is null record;
9
10 procedure Read (Port : in out Serial_Port;
11                Data : out UByte_Array);
12
13 procedure Write (Port : in out Serial_Port;
14                 Data : UByte_Array);
15
16 end Serial_Ports;

```

Listing 2223: serial_ports.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Serial_Ports is
4
5      procedure Display (Data : UByte_Array) is
6      begin
7          Put_Line ("---- Data ----");
8          for E of Data loop
9              Put_Line (UByte'Image (E));
10         end loop;
11         Put_Line ("-----");
12     end Display;
13
14     procedure Read (Port : in out Serial_Port;
15                   Data : out UByte_Array) is
16     pragma Unreferenced (Port);
17     begin
18         Put_Line ("Reading data...");
19         Data := (0, 0, 32, 0);
20     end Read;
21
22     procedure Write (Port : in out Serial_Port;
23                    Data : UByte_Array) is
24     pragma Unreferenced (Port);
25     begin
26         Put_Line ("Writing data...");
27         Display (Data);
28     end Write;
29
30 end Serial_Ports;

```

Listing 2224: data_stream.ads

```

1  with Serial_Ports; use Serial_Ports;
2  with Registers; use Registers;
3
4  package Data_Stream is
5
6      procedure Send (Port : in out Serial_Port;
7                    PMC : PMC_Peripheral);
8
9      procedure Receive (Port : in out Serial_Port;
10                      PMC : out PMC_Peripheral);

```

(continues on next page)

(continued from previous page)

```

11
12 end Data_Stream;

```

Listing 2225: data_stream.adb

```

1 package body Data_Stream is
2
3   procedure Send (Port : in out Serial_Port;
4                  PMC  :      PMC_Peripheral)
5   is
6     Raw_TX : UByte_Array (1 .. PMC'Size / 8)
7       with Address => PMC'Address;
8   begin
9     Write (Port => Port,
10          Data => Raw_TX);
11 end Send;
12
13 procedure Receive (Port : in out Serial_Port;
14                  PMC  :      out PMC_Peripheral)
15 is
16   Raw_TX : UByte_Array (1 .. PMC'Size / 8)
17     with Address => PMC'Address;
18 begin
19   Read (Port => Port,
20        Data => Raw_TX);
21 end Receive;
22
23 end Data_Stream;

```

Listing 2226: test_data_stream.adb

```

1 with Ada.Text_IO;
2
3 with Registers;
4 with Data_Stream;
5 with Serial_Ports;
6
7 procedure Test_Data_Stream is
8
9   procedure Display_Registers is
10     use Ada.Text_IO;
11   begin
12     Put_Line ("---- Registers ----");
13     Put_Line ("PMC_SCER.USBCLK: "
14              & Registers.PMC_Periph.PMC_SCER.USBCLK'Image);
15     Put_Line ("PMC_SCDR.USBCLK: "
16              & Registers.PMC_Periph.PMC_SCDR.USBCLK'Image);
17     Put_Line ("-----");
18   end Display_Registers;
19
20   Port : Serial_Ports.Serial_Port;
21 begin
22   Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
23   Registers.PMC_Periph.PMC_SCDR.USBCLK := 1;
24
25   Display_Registers;
26
27   Data_Stream.Send (Port => Port,
28                    PMC  => Registers.PMC_Periph);
29

```

(continues on next page)

(continued from previous page)

```
30     Data_Stream.Receive (Port => Port,  
31                           PMC  => Registers.PMC_Periph);  
32  
33     Display_Registers;  
34 end Test_Data_Stream;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Embedded.Data_Stream
MD5: 3f4e1a184e52a83b1b9de9e3d5cb43bf

Runtime output

```
---- Registers ----  
PMC_SCER.USBCLK: 1  
PMC_SCDR.USBCLK: 1  
-----  
Writing data...  
---- Data ----  
32  
0  
32  
0  
-----  
Reading data...  
---- Registers ----  
PMC_SCER.USBCLK: 0  
PMC_SCDR.USBCLK: 1  
-----
```

In this example, we can find the overlay in the implementation of the Send and Receive procedures from the `Data_Stream` package. Because the overlay doesn't need to know the internals of the `PMC_Peripheral` type, we're declaring it in the same way as in the previous example (where we created an overlay for `Some_Object`). In this case, we're creating an overlay for the `PMC` parameter.

Note that, for this section, we're not really interested in the details about the serial port. Thus, package `Serial_Ports` in this example is just a stub. However, because the `Serial_Port` type in that package only sees arrays of bytes, after implementing an actual serial port interface for a specific device, we could create data streams for any type.

84.7 ARM and svd2ada

As we've seen in the previous section about *interfacing with devices* (page 1834), Ada offers powerful features to describe low-level details about the hardware architecture without giving up its strong typing capabilities. However, it can be cumbersome to create a specification for all those low-level details when you have a complex architecture. Fortunately, for ARM Cortex-M devices, the GNAT toolchain offers an Ada binding generator called **svd2ada**, which takes CMSIS-SVD descriptions for those devices and creates Ada specifications that match the architecture. CMSIS-SVD description files are based on the Cortex Microcontroller Software Interface Standard (CMSIS), which is a hardware abstraction layer for ARM Cortex microcontrollers.

Please refer to the [svd2ada project page](https://github.com/AdaCore/svd2ada)⁴⁴⁵ for details about this tool.

⁴⁴⁵ <https://github.com/AdaCore/svd2ada>

ENHANCING VERIFICATION WITH SPARK AND ADA

85.1 Understanding Exceptions and Dynamic Checks

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering "exceptions" that interrupt the normal flow of execution. For example, an exception is raised by an attempt to access an array component via an index that is out of bounds. This simple check precludes exploits based on buffer overflow. Several other cases also raise language-defined exceptions, such as scalar range constraint violations and null pointer dereferences. Developers may declare and raise their own application-specific exceptions too. (Exceptions are software artifacts, although an implementation may map hardware events to exceptions.)

Exceptions are raised during execution of what we will loosely define as a "frame." A frame is a language construct that has a call stack entry when called, for example a procedure or function body. There are a few other constructs that are also pertinent but this definition will suffice for now.

Frames have a sequence of statements implementing their functionality. They can also have optional "exception handlers" that specify the response when exceptions are "raised" by those statements. These exceptions could be raised directly within the statements, or indirectly via calls to other procedures and functions.

For example, the frame below is a procedure including three exceptions handlers:

Listing 2227: p.adb

```
1 procedure P is
2 begin
3   Statements_That_Might_Raise_Exceptions;
4 exception
5   when A =>
6     Handle_A;
7   when B =>
8     Handle_B;
9   when C =>
10    Handle_C;
11 end P;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exceptions
MD5: bf7a8740dfca9f3da993f054e22ca97d

The three exception handlers each start with the word **when** (lines 5, 7, and 9). Next comes one or more exception identifiers, followed by the so-called "arrow." In Ada, the arrow always associates something on the left side with something on the right side. In this case, the left side is the exception name and the right side is the handler's code for that exception.

Each handler's code consists of an arbitrary sequence of statements, in this case specific procedures called in response to those specific exceptions. If exception A is raised we call procedure `Handle_A` (line 6), dedicated to doing the actual work of handling that exception. The other two exceptions are dealt with similarly, on lines 8 and 10.

Structurally, the exception handlers are grouped together and textually separated from the rest of the code in a frame. As a result, the sequence of statements representing the normal flow of execution is distinct from the section representing the error handling. The reserved word **exception** separates these two sections (line 4 above). This separation helps simplify the overall flow, increasing understandability. In particular, status result codes are not required so there is no mixture of error checking and normal processing. If no exception is raised the exception handler section is automatically skipped when the frame exits.

Note how the syntactic structure of the exception handling section resembles that of an Ada case statement. The resemblance is intentional, to suggest similar behavior. When something in the statements of the normal execution raises an exception, the corresponding exception handler for that specific exception is executed. After that, the routine completes. The handlers do not "fall through" to the handlers below. For example, if exception B is raised, procedure `Handle_B` is called but `Handle_C` is not called. There's no need for a **break** statement, just as there is no need for it in a case statement. (There's no break statement in Ada anyway.)

So far, we've seen a frame with three specific exceptions handled. What happens if a frame has no handler for the actual exception raised? In that case the run-time library code goes "looking" for one.

Specifically, the active exception is propagated up the dynamic call chain. At each point in the chain, normal execution in that caller is abandoned and the handlers are examined. If that caller has a handler for the exception, the handler is executed. That caller then returns normally to its caller and execution continues from there. Otherwise, propagation goes up one level in the call chain and the process repeats. The search continues until a matching handler is found or no callers remain. If a handler is never found the application terminates abnormally. If the search reaches the main procedure and it has a matching handler it will execute the handler, but, as always, the routine completes so once again the application terminates.

For a concrete example, consider the following:

Listing 2228: arrays.ads

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exceptions
MD5: a2dfa05b56144e21d5796d39c88ceac2

Listing 2229: arrays.adb

```
1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4     begin
5         return A (X + Y * 10);
```

(continues on next page)

(continued from previous page)

```

6   end Value;
7
8   end Arrays;

```

Listing 2230: some_process.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Arrays;      use Arrays;
3
4  procedure Some_Process is
5      L : constant List (1 .. 100) := (others => 42);
6  begin
7      Put_Line (Integer'Image (Value (L, 1, 10)));
8  exception
9      when Constraint_Error =>
10         Put_Line ("Constraint_Error caught in Some_Process");
11         Put_Line ("Some_Process completes normally");
12 end Some_Process;

```

Code block metadata

```

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exceptions
MD5: 7733854601db37eb53f4c4094fe5ca0d

```

Listing 2231: main.adb

```

1  with Some_Process;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Main is
5  begin
6      Some_Process;
7      Put_Line ("Main completes normally");
8  end Main;

```

Procedure Main calls Some_Process, which in turn calls function Value (line 7). Some_Process declares the array object L of type List on line 5, with bounds 1 through 100. The call to Value has arguments, including variable L, leading to an attempt to access an array component via an out-of-bounds index ($1 + 10 * 10 = 101$, beyond the last index of L). This attempt will trigger an exception in Value prior to actually accessing the array object's memory. Function Value doesn't have any exception handlers so the exception is propagated up to the caller Some_Process. Procedure Some_Process has an exception handler for Constraint_Error and it so happens that Constraint_Error is the exception raised in this case. As a result, the code for that handler will be executed, printing some messages on the screen. Then procedure Some_Process will return to Main normally. Main then continues to execute normally after the call to Some_Process and prints its completion message.

If procedure Some_Process had also not had a handler for Constraint_Error, that procedure call would also have returned abnormally and the exception would have been propagated further up the call chain to procedure Main. Normal execution in Main would likewise be abandoned in search of a handler. But Main does not have any handlers so Main would have completed abnormally, immediately, without printing its closing message.

This semantic model is the same as with many other programming languages, in which the execution of a frame's sequence of statements is unavoidably abandoned when an exception becomes active. The model is a direct reaction to the use of status codes returned from functions as in C, where it is all too easy to forget (intentionally or otherwise) to check the status values returned. With the exception model errors cannot be ignored.

However, full exception propagation as described above is not the norm for embedded applications when the highest levels of integrity are required. The run-time library code implementing exception propagation can be rather complex and expensive to certify. Those problems apply to the application code too, because exception propagation is a form of control flow without any explicit construct in the source. Instead of the full exception model, designers of high-integrity applications often take alternative approaches.

One alternative consists of deactivating exceptions altogether, or more precisely, deactivating language-defined checks, which means that the compiler will not generate code checking for conditions giving rise to exceptions. Of course, this makes the code vulnerable to attacks, such as buffer overflow, unless otherwise verified (e.g. through static analysis). Deactivation can be applied at the unit level, through the `-gnatp` compiler switch, or locally within a unit via the pragma `Suppress`. (Refer to the [GNAT User's Guide for Native Platforms](#)⁴⁴⁶ for more details about the switch.)

For example, we can write the following. Note the pragma on line 4 of `arrays.adb` within function `Value`:

Listing 2232: `arrays.ads`

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 2233: `arrays.adb`

```
1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4         pragma Suppress (All_Checks);
5     begin
6         return A (X + Y * 10);
7     end Value;
8
9 end Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exception_Suppress
MD5: 62c37774cbcd5f167858d3b5268006aa

Listing 2234: `some_process.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5     L : constant List (1 .. 100) := (others => 42);
6 begin
7     Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9     when Constraint_Error =>
10         Put_Line ("FAILURE");
11 end Some_Process;
```

This placement of the pragma will only suppress checks in the function body. However,

⁴⁴⁶ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html

that is where the exception would otherwise have been raised, leading to incorrect and unpredictable execution. (Run the program more than once. If it prints the right answer (42), or even the same value each time, it's just a coincidence.) As you can see, suppressing checks negates the guarantee of errors being detected and addressed at run-time.

Another alternative is to leave checks enabled but not retain the dynamic call-chain propagation. There are a couple of approaches available in this alternative.

The first approach is for the run-time library to invoke a global "last chance handler" (LCH) when any exception is raised. Instead of the sequence of statements of an ordinary exception handler, the LCH is actually a procedure intended to perform "last-wishes" before the program terminates. No exception handlers are allowed. In this scheme "propagation" is simply a direct call to the LCH procedure. The default LCH implementation provided by GNAT does nothing other than loop infinitely. Users may define their own replacement implementation.

The availability of this approach depends on the run-time library. Typically, *Zero Footprint* and *Ravenscar SFP* run-times will provide this mechanism because they are intended for certification.

A user-defined LCH handler can be provided either in C or in Ada, with the following profiles:

[Ada]

```
procedure Last_Chance_Handler (Source_Location : System.Address; Line : Integer);
pragma Export (C,
               Last_Chance_Handler,
               "__gnat_last_chance_handler");
```

[C]

```
void __gnat_last_chance_handler (char *source_location,
                                int line);
```

We'll go into the details of the pragma Export in a further section on language interfacing. For now, just know that the symbol `__gnat_last_chance_handler` is what the run-time uses to branch immediately to the last-chance handler. Pragma Export associates that symbol with this replacement procedure so it will be invoked instead of the default routine. As a consequence, the actual procedure name in Ada is immaterial.

Here is an example implementation that simply blinks an LED forever on the target:

```
procedure Last_Chance_Handler (Msg : System.Address; Line : Integer) is
  pragma Unreferenced (Msg, Line);

  Next_Release : Time := Clock;
  Period       : constant Time_Span := Milliseconds (500);
begin
  Initialize_LEDs;
  All_LEDs_Off;

  loop
    Toggle (LCH_LED);
    Next_Release := Next_Release + Period;
    delay until Next_Release;
  end loop;
end Last_Chance_Handler;
```

The `LCH_LED` is a constant referencing the LED used by the last-chance handler, declared elsewhere. The infinite loop is necessary because a last-chance handler must never return to the caller (hence the term "last-chance"). The LED changes state every half-second.

Unlike the approach in which there is only the last-chance handler routine, the other approach allows exception handlers, but in a specific, restricted manner. Whenever an ex-

ception is raised, the only handler that can apply is a matching handler located in the same frame in which the exception is raised. Propagation in this context is simply an immediate branch instruction issued by the compiler, going directly to the matching handler's sequence of statements. If there is no matching local handler the last chance handler is invoked. For example consider the body of function `Value` in the body of package `Arrays`:

Listing 2235: `arrays.ads`

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 2236: `arrays.adb`

```
1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4     begin
5         return A (X + Y * 10);
6     exception
7         when Constraint_Error =>
8             return 0;
9     end Value;
10
11 end Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exception_Return
MD5: 1f63b92739deb03529884ab0d25dadb8

Listing 2237: `some_process.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5     L : constant List (1 .. 100) := (others => 42);
6 begin
7     Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9     when Constraint_Error =>
10         Put_Line ("FAILURE");
11 end Some_Process;
```

In both procedure `Some_Process` and function `Value` we have an exception handler for `Constraint_Error`. In this example the exception is raised in `Value` because the index check fails there. A local handler for that exception is present so the handler applies and the function returns zero, normally. Because the call to the function returns normally, the execution of `Some_Process` prints zero and then completes normally.

Let's imagine, however, that function `Value` did *not* have a handler for `Constraint_Error`. In the context of full exception propagation, the function call would return to the caller, i.e., `Some_Process`, and would be handled in that procedure's handler. But only local handlers are allowed under the second alternative so the lack of a local handler in `Value` would result in the last-chance handler being invoked. The handler for `Constraint_Error` in `Some_Process` under this alternative approach.

So far we've only illustrated handling the `Constraint_Error` exception. It's possible to handle other language-defined and user-defined exceptions as well, of course. It is even possible to define a single handler for all other exceptions that might be encountered in the handled sequence of statements, beyond those explicitly named. The "name" for this otherwise anonymous exception is the Ada reserved word **others**. As in case statements, it covers all other choices not explicitly mentioned, and so must come last. For example:

Listing 2238: arrays.ads

```

1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 2239: arrays.adb

```

1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4     begin
5         return A (X + Y * 10);
6     exception
7         when Constraint_Error =>
8             return 0;
9         when others =>
10            return -1;
11     end Value;
12
13 end Arrays;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Exception_Return_Others
MD5: 7c2ed7efa23242f502a6cf4767da0192

Listing 2240: some_process.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5     L : constant List (1 .. 100) := (others => 42);
6 begin
7     Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9     when Constraint_Error =>
10        Put_Line ("FAILURE");
11 end Some_Process;
```

In the code above, the `Value` function has a handler specifically for `Constraint_Error` as before, but also now has a handler for all other exceptions. For any exception other than `Constraint_Error`, function `Value` returns -1. If you remove the function's handler for `Constraint_Error` (lines 7 and 8) then the other "anonymous" handler will catch the exception and -1 will be returned instead of zero.

There are additional capabilities for exceptions, but for now you have a good basic understanding of how exceptions work, especially their dynamic nature at run-time.

85.2 Understanding Dynamic Checks versus Formal Proof

So far, we have discussed language-defined checks inserted by the compiler for verification at run-time, leading to exceptions being raised. We saw that these dynamic checks verified semantic conditions ensuring proper execution, such as preventing writing past the end of a buffer, or exceeding an application-specific integer range constraint, and so on. These checks are defined by the language because they apply generally and can be expressed in language-defined terms.

Developers can also define dynamic checks. These checks specify component-specific or application-specific conditions, expressed in terms defined by the component or application. We will refer to these checks as "user-defined" for convenience. (Be sure you understand that we are not talking about user-defined *exceptions* here.)

Like the language-defined checks, user-defined checks must be true at run-time. All checks consist of Boolean conditions, which is why we can refer to them as assertions: their conditions are asserted to be true by the compiler or developer.

Assertions come in several forms, some relatively low-level, such as a simple pragma `Assert`, and some high-level, such as type invariants and contracts. These forms will be presented in detail in a later section, but we will illustrate some of them here.

User-defined checks can be enabled at run-time in GNAT with the `-gnata` switch, as well as with pragma `Assertion_Policy`. The switch enables all forms of these assertions, whereas the pragma can be used to control specific forms. The switch is typically used but there are reasonable use-cases in which some user-defined checks are enabled, and others, although defined, are disabled.

By default in GNAT, language-defined checks are enabled but user-defined checks are disabled. Here's an example of a simple program employing a low-level assertion. We can use it to show the effects of the switches, including the defaults:

Listing 2241: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   X : Positive := 10;
5 begin
6   X := X * 5;
7   pragma Assert (X > 99);
8   X := X - 99;
9   Put_Line (Integer'Image (X));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Low_Level_Assertion
MD5: 2eb5e1879740cc3914acb8a362995b31

If we compiled this code we would get a warning about the assignment on line 8 after the pragma `Assert`, but not one about the `Assert` itself on line 7.

```

gprbuild -q -P main.gpr
main.adb:8:11: warning: value not in range of type "Standard.Positive"
main.adb:8:11: warning: "Constraint_Error" will be raised at run time
```

No code is generated for the user-defined check expressed via pragma `Assert` but the language-defined check is emitted. In this case the range constraint on `X` excludes zero and negative numbers, but $X * 5 = 50$, $X - 99 = -49$. As a result, the check for the last

assignment would fail, raising `Constraint_Error` when the program runs. These results are the expected behavior for the default switch settings.

But now let's enable user-defined checks and build it. Different compiler output will appear.

Listing 2242: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   X : Positive := 10;
5 begin
6   X := X * 5;
7   pragma Assert (X > 99);
8   X := X - 99;
9   Put_Line (Integer'Image (X));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Assert
MD5: 2eb5e1879740cc3914acb8a362995b31

Build output

```

main.adb:7:19: warning: assertion will fail at run time [-gnatw.a]
main.adb:8:11: warning: value not in range of type "Standard.Positive" [enabled by ↵
↵default]
main.adb:8:11: warning: Constraint_Error will be raised at run time [enabled by ↵
↵default]
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : main.adb:7
```

Now we also get the compiler warning about the pragma Assert condition. When run, the failure of pragma Assert on line 7 raises the exception `Ada.Assertions.Assertion_Error`. According to the expression in the assertion, `X` is expected (incorrectly) to be above 99 after the multiplication. (The exception name in the error message, `SYSTEM ASSERTIONS ASSERT_FAILURE`, is a GNAT-specific alias for `Ada.Assertions.Assertion_Error`.)

It's interesting to see in the output that the compiler can detect some violations at compile-time:

```

main.adb:7:19: warning: assertion will fail at run time
main.adb:7:21: warning: condition can only be True if invalid values present
main.adb:8:11: warning: value not in range of type "Standard.Positive"
```

Generally speaking, a complete analysis is beyond the scope of compilers and they may not find all errors prior to execution, even those we might detect ourselves by inspection. More errors can be found by tools dedicated to that purpose, known as static analyzers. But even an automated static analysis tool cannot guarantee it will find all potential problems.

A much more powerful alternative is formal proof, a form of static analysis that can (when possible) give strong guarantees about the checks, for all possible conditions and all possible inputs. Proof can be applied to both language-defined and user-defined checks.

Be sure you understand that formal proof, as a form of static analysis, verifies conditions prior to execution, even prior to compilation. That earliness provides significant cost benefits. Removing bugs earlier is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after

deployment. Preventing bug introduction into the deployed system is the least expensive approach of all. Furthermore, cost savings during the initial development will be possible as well, for reasons specific to proof. We will revisit this topic later in this section.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **gnatprove** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof. As a disclaimer, this course is not aimed at providing a full introduction to proof and the SPARK language, but rather to present in a few examples what it is about and what it can do for us.

As it turns out, our procedure Main is already SPARK compliant so we can start verifying it.

Listing 2243: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   X : Positive := 10;
5 begin
6   X := X * 5;
7   pragma Assert (X > 99);
8   X := X - 99;
9   Put_Line (Integer'Image (X));
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Assert
MD5: 98cad2c7e7b7a12740db013727f01d45

Build output

```
main.adb:7:20: warning: assertion will fail at run time [-gnatw.a]
main.adb:8:12: warning: value not in range of type "Standard.Positive" [enabled by ↵
↵default]
main.adb:8:12: warning: Constraint_Error will be raised at run time [enabled by ↵
↵default]
```

Prover output

```
Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
main.adb:7:20: medium: assertion might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA.ASSERTIONS.ASSERTION_ERROR : main.adb:7
```

The "Prove" button invokes **gnatprove** on main.adb. You can ignore the parameters to the invocation. For the purpose of this demonstration, the interesting output is this message:

```
main.adb:7:19: medium: assertion might fail, cannot prove X > 99 (e.g. when X = 50)
```

gnatprove can tell that the assertion `X > 99` may have a problem. There's indeed a bug here, and **gnatprove** even gives us the counterexample (when X is 50). As a result the code is not proven and we know we have an error to correct.

Notice that the message says the assertion "might fail" even though clearly **gnatprove** has an example for when failure is certain. That wording is a reflection of the fact that SPARK gives strong guarantees when the assertions are proven to hold, but does not guarantee that flagged problems are indeed problems. In other words, **gnatprove** does not give false positives but false negatives are possible. The result is that if **gnatprove** does not indicate a problem for the code under analysis we can be sure there is no problem, but if **gnatprove** does indicate a problem the tool may be wrong.

85.3 Initialization and Correct Data Flow

An immediate benefit from having our code compatible with the SPARK subset is that we can ask **gnatprove** to verify initialization and correct data flow, as indicated by the absence of messages during SPARK "flow analysis." Flow analysis detects programming errors such as reading uninitialized data, problematic aliasing between formal parameters, and data races between concurrent tasks.

In addition, **gnatprove** checks unit specifications for the actual data read or written, and the flow of information from inputs to outputs. As you can imagine, this verification provides significant benefits, and it can be reached with comparatively low cost.

For example, the following illustrates an initialization failure:

Listing 2244: main.adb

```

1 with Increment;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Main is
5   B : Integer;
6 begin
7   Increment (B);
8   Put_Line (B'Image);
9 end Main;
```

Listing 2245: increment.adb

```

1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_0
MD5: 06d432a84d94635bb7bddafd9574a748

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
main.adb:7:15: high: "B" is not initialized
gnatprove: unproved check messages considered as errors
```

Granted, Increment is a silly procedure as-is, but imagine it did useful things, and, as part of that, incremented the argument. **gnatprove** tells us that the caller has not assigned a value to the argument passed to Increment.

Consider this next routine, which contains a serious coding error. Flow analysis will find it for us.

Listing 2246: compute_offset.adb

```

1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 procedure Compute_Offset (K : Float; Z : out Integer; Flag : out Boolean) is
4   X : constant Float := Sin (K);
5 begin
6   if X < 0.0 then
7     Z := 0;
8     Flag := True;
9   elsif X > 0.0 then
10    Z := 1;
11    Flag := True;
12   else
13     Flag := False;
14   end if;
15 end Compute_Offset;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_1
MD5: af7f16a9c83359c49fde44ed4796c8ec

Prover output

```

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: analysis of data and information flow ...
compute_offset.adb:3:38: medium: "Z" might not be initialized in "Compute_Offset"
↳[reason for check: OUT parameter should be initialized on return] [possible fix:
↳initialize "Z" on all paths or make "Z" an IN OUT parameter]
gnatprove: unproved check messages considered as errors

```

gnatprove tells us that *Z* might not be initialized (assigned a value) in *Compute_Offset*, and indeed that is correct. *Z* is a mode **out** parameter so the routine should assign a value to it: *Z* is an output, after all. The fact that *Compute_Offset* does not do so is a significant and nasty bug. Why is it so nasty? In this case, formal parameter *Z* is of the scalar type **Integer**, and scalar parameters are always passed by copy in Ada and SPARK. That means that, when returning to the caller, an integer value is copied to the caller's argument passed to *Z*. But this procedure doesn't always assign the value to be copied back, and in that case an arbitrary value — whatever is on the stack — is copied to the caller's argument. The poor programmer must debug the code to find the problem, yet the effect could appear well downstream from the call to *Compute_Offset*. That's not only painful, it is expensive. Better to find the problem before we even compile the code.

85.4 Contract-Based Programming

So far, we've seen assertions in a routine's sequence of statements, either through implicit language-defined checks (is the index in the right range?) or explicit user-defined checks. These checks are already useful by themselves but they have an important limitation: the assertions are in the implementation, hidden from the callers of the routine. For example, a call's success or failure may depend upon certain input values but the caller doesn't have that information.

Generally speaking, Ada and SPARK put a lot of emphasis on strong, complete specifications for the sake of abstraction and analysis. Callers need not examine the implementations to determine whether the arguments passed to it are changed, for example. It is possible to go beyond that, however, to specify implementation constraints and functional requirements. We use contracts to do so.

At the language level, contracts are higher-level forms of assertions associated with specifications and declarations rather than sequences of statements. Like other assertions they can be activated or deactivated at run-time, and can be statically proven. We'll concentrate here on two kinds of contracts, both associated especially (but not exclusively) with procedures and functions:

- *Preconditions*, those Boolean conditions required to be true *prior* to a call of the corresponding subprogram
- *Postconditions*, those Boolean conditions required to be true *after* a call, as a result of the corresponding subprogram's execution

In particular, preconditions specify the initial conditions, if any, required for the called routine to correctly execute. Postconditions, on the other hand, specify what the called routine's execution must have done, at least, on normal completion. Therefore, preconditions are obligations on callers (referred to as "clients") and postconditions are obligations on implementers. By the same token, preconditions are guarantees to the implementers, and postconditions are guarantees to clients.

Contract-based programming, then, is the specification and rigorous enforcement of these obligations and guarantees. Enforcement is rigorous because it is not manual, but tool-based: dynamically at run-time with exceptions, or, with SPARK, statically, prior to build.

Preconditions are specified via the "Pre" aspect. Postconditions are specified via the "Post" aspect. Usually subprograms have separate declarations and these aspects appear with those declarations, even though they are *about* the bodies. Placement on the declarations allows the obligations and guarantees to be visible to all parties. For example:

Listing 2247: mid.ads

```

1 function Mid (X, Y : Integer) return Integer with
2   Pre  => X + Y /= 0,
3   Post => Mid'Result > X;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_2
MD5: 0fb78847a167d9318b00667c59a7038d

The precondition on line 2 specifies that, for any given call, the sum of the values passed to parameters X and Y must not be zero. (Perhaps we're dividing by X + Y in the body.) The declaration also provides a guarantee about the function call's result, via the postcondition on line 3: for any given call, the value returned will be greater than the value passed to X.

Consider a client calling this function:

Listing 2248: demo.adb

```

1 with Mid;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Demo is
5   A, B, C : Integer;
6 begin
7   A := Mid (1, 2);
8   B := Mid (1, -1);
9   C := Mid (A, B);
10  Put_Line (C'Image);
11 end Demo;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_2
MD5: 3e0617d4b1c14b37a81377456bf73eb5

Prover output

```
Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
demo.adb:8:09: medium: precondition might fail
gnatprove: unproved check messages considered as errors
```

gnatprove indicates that the assignment to B (line 8) might fail because of the precondition, i.e., the sum of the inputs shouldn't be 0, yet $-1 + 1 = 0$. (We will address the other output message elsewhere.)

Let's change the argument passed to Y in the second call (line 8). Instead of -1 we will pass -2:

Listing 2249: demo.adb

```
1 with Mid;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Demo is
5   A, B, C : Integer;
6 begin
7   A := Mid (1, 2);
8   B := Mid (1, -2);
9   C := Mid (A, B);
10  Put_Line (C'Image);
11 end Demo;
```

Listing 2250: mid.ads

```
1 function Mid (X, Y : Integer) return Integer with
2   Pre  => X + Y /= 0,
3   Post => Mid'Result > X;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_3
MD5: 496937d76e16ba524f98f5a94398e929

Prover output

```
Phase 1 of 3: generation of data representation information ...
generation of data representation information failed
continuing analysis with partial data representation
for details, see log file gnatprove/data_representation_generation.log
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
```

The second call will no longer be flagged for the precondition. In addition, **gnatprove** will know from the postcondition that A has to be greater than 1, as does B, because in both calls 1 was passed to X. Therefore, **gnatprove** can deduce that the precondition will hold for the third call `C := Mid (A, B)`; because the sum of two numbers greater than 1 will never be zero.

Postconditions can also compare the state prior to a call with the state after a call, using

the `'Old` attribute. For example:

Listing 2251: increment.ads

```
1 procedure Increment (Value : in out Integer) with
2   Pre => Value < Integer'Last,
3   Post => Value = Value'Old + 1;
```

Listing 2252: increment.adb

```
1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_4
MD5: b879dcff91cb4fbce5501474b7f2e732

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

The postcondition specifies that, on return, the argument passed to the parameter `Value` will be one greater than it was immediately prior to the call (`Value'Old`).

85.5 Replacing Defensive Code

One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations. For example, the *Push* operation for a stack type would need to ensure that the given stack is not already full. The body of the routine would first check that, explicitly, and perhaps raise an exception or set a status code. With preconditions we can make the requirement explicit and **gnatprove** will verify that the requirement holds at all call sites.

This reduction has a number of advantages:

- The implementation is simpler, removing validation code that is often difficult to test, makes the code more complex and leads to behaviors that are difficult to define.
- The precondition documents the conditions under which it's correct to call the subprogram, moving from an implementer responsibility to mitigate invalid input to a user responsibility to fulfill the expected interface.
- Provides the means to verify that this interface is properly respected, through code review, dynamic checking at run-time, or formal static proof.

As an example, consider a procedure `Read` that returns a component value from an array. Both the `Data` and `Index` are objects visible to the procedure so they are not formal parameters.

Listing 2253: p.ads

```
1 package P is
2
3   type List is array (Integer range <>) of Character;
4
5   Data : List (1 .. 100);
```

(continues on next page)

(continued from previous page)

```

6   Index : Integer := Data'First;
7
8   procedure Read (V : out Character);
9
10  end P;

```

Listing 2254: p.adb

```

1  package body P is
2
3      procedure Read (V : out Character) is
4      begin
5          if Index not in Data'Range then
6              V := Character'First;
7              return;
8          end if;
9
10         V := Data (Index);
11         Index := Index + 1;
12     end Read;
13 end P;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Defensive
MD5: 4b4767100079b228f4f3c630d267ec53

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

In addition to procedure Read we would also have a way to load the array components in the first place, but we can ignore that for the purpose of this discussion.

Procedure Read is responsible for reading an element of the array and then incrementing the index. What should it do in case of an invalid index? In this implementation there is defensive code that returns a value arbitrarily chosen. We could also redesign the code to return a status in this case, or — better — raise an exception.

An even more robust approach would be instead to ensure that this subprogram is only called when Index is within the indexing boundaries of Data. We can express that requirement with a precondition (line 9).

Listing 2255: p.ads

```

1  package P is
2
3      type List is array (Integer range <>) of Character;
4
5      Data : List (1 .. 100);
6      Index : Integer := 1;
7
8      procedure Read (V : out Character)
9          with Pre => Index in Data'Range;
10
11  end P;

```

Listing 2256: p.adb

```

1 package body P is
2
3     procedure Read (V : out Character) is
4     begin
5         V := Data (Index);
6         Index := Index + 1;
7     end Read;
8
9 end P;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Defensive
MD5: 9646614c34d191be51b4522c972538aa

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

Now we don't need the defensive code in the procedure body. That's safe because SPARK will attempt to prove statically that the check will not fail at the point of each call.

Assuming that procedure Read is intended to be the only way to get values from the array, in a real application (where the principles of software engineering apply) we would take advantage of the compile-time visibility controls that packages offer. Specifically, we would move all the variables' declarations to the private part of the package, or even the package body, so that client code could not possibly access the array directly. Only procedure Read would remain visible to clients, thus remaining the only means of accessing the array. However, that change would entail others, and in this chapter we are only concerned with introducing the capabilities of SPARK. Therefore, we keep the examples as simple as possible.

85.6 Proving Absence of Run-Time Errors

Earlier we said that **gnatprove** will verify both language-defined and user-defined checks. Proving that the language-defined checks will not raise exceptions at run-time is known as proving "Absence of Run-Time Errors" or AoRTE for short. Successful proof of these checks is highly significant in itself.

One of the major resulting benefits is that we can deploy the final executable with checks disabled. That has obvious performance benefits, but it is also a safety issue. If we disable the checks we also disable the run-time library support for them, but in that case the language does not define what happens if indeed an exception is raised. Formally speaking, anything could happen. We must have good reason for thinking that exceptions cannot be raised.

This is such an important issue that proof of AoRTE can be used to comply with the objectives of certification standards in various high-integrity domains (for example, DO-178B/C in avionics, EN 50128 in railway, IEC 61508 in many safety-related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, and ISO 26262 in automotive).

As a result, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in other programming languages.

However, successful proof of AoRTE may require additional assertions, especially preconditions. We can see that with procedure Increment, the procedure that takes an Integer

argument and increments it by one. But of course, if the incoming value of the argument is the largest possible positive value, the attempt to increment it would overflow, raising `Constraint_Error`. (As you have likely already concluded, `Constraint_Error` is the most common exception you will have to deal with.) We added a precondition to allow only the integer values up to, but not including, the largest positive value:

Listing 2257: increment.ads

```
1 procedure Increment (Value : in out Integer) with
2   Pre => Value < Integer'Last,
3   Post => Value = Value'Old + 1;
```

Listing 2258: increment.adb

```
1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.SPARK.Contracts_5
MD5: b879dcff91cb4fbce5501474b7f2e732

Prover output

Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

Prove it, then comment-out the precondition and try proving it again. Not only will **gnat-prove** tell us what is wrong, it will suggest a solution as well.

Without the precondition the check it provides would have to be implemented as defensive code in the body. One or the other is critical here, but note that we should never need both.

85.7 Proving Abstract Properties

The postcondition on `Increment` expresses what is, in fact, a unit-level requirement. Successfully proving such requirements is another significant robustness and cost benefit. Together with the proofs for initialization and AoRTE, these proofs ensure program integrity, that is, the program executes within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors, and data cannot be corrupted.

We can go even further. We can use contracts to express arbitrary abstract properties when such exist. Safety and security properties, for instance, could be expressed as postconditions and then proven by **gnatprove**.

For example, imagine we have a procedure to move a train to a new position on the track, and we want to do so safely, without leading to a collision with another train. Procedure `Move`, therefore, takes two inputs: a train identifier specifying which train to move, and the intended new position. The procedure's output is a value indicating a motion command to be given to the train in order to go to that new position. If the train cannot go to that new position safely the output command is to stop the train. Otherwise the command is for the train to continue at an indicated speed:

```
type Move_Result is (Full_Speed, Slow_Down, Keep_Going, Stop);

procedure Move
```

(continues on next page)

(continued from previous page)

```

    (Train      : in Train_Id;
     New_Position : in Train_Position;
     Result      : out Move_Result)
with
  Pre => Valid_Id (Train) and
        Valid_Move (Trains (Train), New_Position) and
        At_Most_One_Train_Per_Track and
        Safe_Signaling,
  Post => At_Most_One_Train_Per_Track and
        Safe_Signaling;

function At_Most_One_Train_Per_Track return Boolean;

function Safe_Signaling return Boolean;

```

The preconditions specify that, given a safe initial state and a valid move, the result of the call will also be a safe state: there will be at most one train per track section and the track signaling system will not allow any unsafe movements.

85.8 Final Comments

Make sure you understand that **gnatprove** does not attempt to prove the program correct as a whole. It attempts to prove language-defined and user-defined assertions about parts of the program, especially individual routines and calls to those routines. Furthermore, **gnatprove** proves the routines correct only to the extent that the user-defined assertions correctly and sufficiently describe and constrain the implementation of the corresponding routines.

Although we are not proving whole program correctness, as you will have seen — and done — we can prove properties that make our software far more robust and bug-free than is possible otherwise. But in addition, consider what proving the unit-level requirements for your procedures and functions would do for the cost of unit testing and system integration. The tests would pass the first time.

However, within the scope of what SPARK can do, not everything can be proven. In some cases that is because the software behavior is not amenable to expression as boolean conditions (for example, a mouse driver). In other cases the source code is beyond the capabilities of the analyzers that actually do the mathematical proof. In these cases the combination of proof and actual test is appropriate, and still less expensive than testing alone.

There is, of course, much more to be said about what can be done with SPARK and **gnatprove**. Those topics are reserved for the [Introduction to SPARK](#) (page 1411) course.

C TO ADA TRANSLATION PATTERNS

86.1 Naming conventions and casing considerations

One question that may arise relatively soon when converting from C to Ada is the style of source code presentation. The Ada language doesn't impose any particular style and for many reasons, it may seem attractive to keep a C-like style — for example, camel casing — to the Ada program.

However, the code in the Ada language standard, most third-party code, and the libraries provided by GNAT follow a specific style for identifiers and reserved words. Using a different style for the rest of the program leads to inconsistencies, thereby decreasing readability and confusing automatic style checkers. For those reasons, it's usually advisable to adopt the Ada style — in which each identifier starts with an upper case letter, followed by lower case letters (or digits), with an underscore separating two "distinct" words within the identifier. Acronyms within identifiers are in upper case. For example, there is a language-defined package named `Ada.Text_IO`. Reserved words are all lower case.

Following this scheme doesn't preclude adding additional, project-specific rules.

86.2 Manually interfacing C and Ada

Before even considering translating code from C to Ada, it's worthwhile to evaluate the possibility of keeping a portion of the C code intact, and only translating selected modules to Ada. This is a necessary evil when introducing Ada to an existing large C codebase, where re-writing the entire code upfront is not practical nor cost-effective.

Fortunately, Ada has a dedicated set of features for interfacing with other languages. The Interfaces package hierarchy and the pragmas `Convention`, `Import`, and `Export` allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

[C]

Listing 2259: call.c

```
1  #include <stdio.h>
2
3  struct my_struct {
4      int A, B;
5  };
6
7  void call (struct my_struct *p) {
8      printf ("%d", p->A);
9  }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.My_Struct_C
MD5: 67053ec329fa4dfc8d8d6125589b9fcb

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C **struct my_struct** is represented, we can use the following record along with a **pragma Convention**. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

[Ada]

Listing 2260: use_my_struct.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Interfaces.C;
3
4 procedure Use_My_Struct is
5
6     type my_struct is record
7         A : Interfaces.C.int;
8         B : Interfaces.C.int;
9     end record;
10    pragma Convention (C, my_struct);
11
12    V : my_struct := (A => 1, B => 2);
13 begin
14     Put_Line ("V = ( "
15              & Interfaces.C.int'Image (V.A)
16              & Interfaces.C.int'Image (V.B)
17              & ")");
18 end Use_My_Struct;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.My_Struct_Ada
MD5: d19942018679df6fbab99f1c6bfdebc8

Runtime output

```
V = ( 1 2)
```

Describing a foreign subprogram call to Ada code is called *binding* and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a void function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a **pragma Import**:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The Import pragma specifies that whenever Call is invoked by Ada code, it should invoke the Call function with the C calling convention.

And that's all that's necessary. Here's an example of a call to Call:

[Ada]

Listing 2261: use_my_struct.adb

```
1 with Interfaces.C;
2
3 procedure Use_My_Struct is
```

(continues on next page)

(continued from previous page)

```

4
5  type my_struct is record
6      A : Interfaces.C.int;
7      B : Interfaces.C.int;
8  end record;
9  pragma Convention (C, my_struct);
10
11  procedure Call (V : my_struct);
12  pragma Import (C, Call, "call"); -- Third argument optional
13
14  V : my_struct := (A => 1, B => 2);
15  begin
16      Call (V);
17  end Use_My_Struct;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.My_Struct
MD5: 9b54edadd406c7f5a2b9f8b8f82a4a88

86.3 Building and Debugging mixed language code

The easiest way to build an application using mixed C / Ada code is to create a simple project file for **gprbuild** and specify C as an additional language. By default, when using **gprbuild** we only compile Ada source files. To compile C code files as well, we use the `Languages` attribute and specify `c` as an option, as in the following example of a project file named *default.gpr*:

```

project Default is
    for Languages use ("ada", "c");
    for Main use ("main.adb");
end Default;

```

Then, we use this project file to build the application by simply calling **gprbuild**. Alternatively, we can specify the project file on the command-line with the `-P` option — for example, `gprbuild -P default.gpr`. In both cases, **gprbuild** compiles all C source-code file found in the directory and links the corresponding object files to build the executable.

In order to include debug information, you can use `gprbuild -cargs -g`. This option adds debug information based on both C and Ada code to the executable. Alternatively, you can specify a `Builder` package in the project file and include global compilation switches for each language using the `Global_Compilation_Switches` attribute. For example:

```

project Default is
    for Languages use ("ada", "c");
    for Main use ("main.adb");

    package Builder is
        for Global_Compilation_Switches ("Ada") use ("-g");
        for Global_Compilation_Switches ("C") use ("-g");
    end Builder;
end Default;

```

In this case, you can simply run `gprbuild -P default.gpr` to build the executable.

To debug the executable, you can use programs such as **gdb** or **ddd**, which are suitable for debugging both C and Ada source-code. If you prefer a complete IDE, you may want to look into **GNAT Studio**, which supports building and debugging an application within a single environment, and remotely running applications loaded to various embedded devices. You can find more information about **gprbuild** and **GNAT Studio** in the [Introduction to GNAT Toolchain](#) (page 2127) course.

86.4 Automatic interfacing

It may be useful to start interfacing Ada and C by using automatic binding generators. These can be done either by invoking **gcc -fdump-ada-spec** option (to generate an Ada binding to a C header file) or **-gnatceg** option (to generate a C binding to an Ada specification file). For example:

```
gcc -c -fdump-ada-spec my_header.h
gcc -c -gnatceg spec.ads
```

The level of interfacing is very low level and typically requires either massaging (changing the generated files) or wrapping (calling the generated files from a higher level interface). For example, numbers bound from C to Ada are only standard numbers where user-defined types may be desirable. C uses a lot of by-pointer parameters which may be better replaced by other parameter modes, etc.

However, the automatic binding generator helps having a starting point which ensures compatibility of the Ada and the C code.

86.5 Using Arrays in C interfaces

It is relatively straightforward to pass an array from Ada to C. In particular, with the GNAT compiler, passing an array is equivalent to passing a pointer to its first element. Of course, as there's no notion of boundaries in C, the length of the array needs to be passed explicitly. For example:

[C]

Listing 2262: p.h

```
1 void p (int * a, int length);
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_1
MD5: 123353e301a3d43016d2799855e6732a

[Ada]

Listing 2263: main.adb

```
1 procedure Main is
2   type Arr is array (Integer range <>) of Integer;
3
4   procedure P (V : Arr; Length : Integer);
5   pragma Import (C, P);
6
7   X : Arr (5 .. 15);
8 begin
9   P (X, X'Length);
10 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_1
 MD5: 9bfb0f31da4554a1e1dea1ba2b1d305

The other way around — that is, retrieving an array that has been created on the C side — is more difficult. Because C doesn't explicitly carry boundaries, they need to be recreated in some way.

The first option is to actually create an Ada array without boundaries. This is the most flexible, but also the least safe option. It involves creating an array with indices over the full range of **Integer** without ever creating it from Ada, but instead retrieving it as an access from C. For example:

[C]

Listing 2264: f.h

```
1 int * f ();
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_2
 MD5: 19e33efb6d7d46778b88baa2709111e5

[Ada]

Listing 2265: main.adb

```
1 procedure Main is
2   type Arr is array (Integer) of Integer;
3   type Arr_A is access all Arr;
4
5   function F return Arr_A;
6   pragma Import (C, F);
7 begin
8   null;
9 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_2
 MD5: b52213bcdd8db5e8abfcb8effabb84df

Note that Arr is a constrained type (it doesn't have the **range** <> notation for indices). For that reason, as it would be for C, it's possible to iterate over the whole range of integer, beyond the memory actually allocated for the array.

A somewhat safer way is to overlay an Ada array over the C one. This requires having access to the length of the array. This time, let's consider two cases, one with an array and its size accessible through functions, another one on global variables. This time, as we're using an overlay, the function will be directly mapped to an Ada function returning an address:

[C]

Listing 2266: fg.h

```
1 int * f_arr (void);
2 int f_size (void);
3
4 int * g_arr;
5 int g_size;
```


Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_3
MD5: b315ec2e5d9fdd297ba295ccbae910bc

[Ada]

Listing 2267: fg.ads

```
1 with System;
2
3 package Fg is
4
5     type Arr is array (Integer range <>) of Integer;
6
7     function F_Arr return System.Address;
8     pragma Import (C, F_Arr, "f_arr");
9
10    function F_Size return Integer;
11    pragma Import (C, F_Size, "f_size");
12
13    F : Arr (0 .. F_Size - 1) with Address => F_Arr;
14
15    G_Size : Integer;
16    pragma Import (C, G_Size, "g_size");
17
18    G_Arr : Arr (0 .. G_Size - 1);
19    pragma Import (C, G_Arr, "g_arr");
20
21 end Fg;
```

Listing 2268: main.adb

```
1 with Fg;
2
3 procedure Main is
4 begin
5     null;
6 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Arr_3
MD5: 5c74f9bca93520ecf85a2010760cc2f8

With all solutions though, importing an array from C is a relatively unsafe pattern, as there's only so much information on the array as there would be on the C side in the first place. These are good places for careful peer reviews.

86.6 By-value vs. by-reference types

When interfacing Ada and C, the rules of parameter passing are a bit different with regards to what's a reference and what's a copy. Scalar types and pointers are passed by value, whereas record and arrays are (almost) always passed by reference. However, there may be cases where the C interface also passes values and not pointers to objects. Here's a slightly modified version of a previous example to illustrate this point:

[C]

Listing 2269: call.c

```

1  #include <stdio.h>
2
3  struct my_struct {
4      int A, B;
5  };
6
7  void call (struct my_struct p) {
8      printf ("%d", p.A);
9  }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Param_By_Value_C
MD5: 42b6e329c5dbfcae368078ca7635341f

In Ada, a type can be modified so that parameters of this type can always be passed by copy.

[Ada]

Listing 2270: main.adb

```

1  with Interfaces.C;
2
3  procedure Main is
4      type my_struct is record
5          A : Interfaces.C.int;
6          B : Interfaces.C.int;
7      end record
8      with Convention => C_Pass_By_Copy;
9
10     procedure Call (V : my_struct);
11     pragma Import (C, Call, "call");
12 begin
13     null;
14 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Param_By_Value_Ada
MD5: 16e97033bdfb2bacc0cf3322c019a94

Note that this cannot be done at the subprogram declaration level, so if there is a mix of by-copy and by-reference calls, two different types need to be used on the Ada side.

86.7 Naming and prefixes

Because of the absence of namespaces, any global name in C tends to be very long. And because of the absence of overloading, they can even encode type names in their type.

In Ada, the package is a namespace — two entities declared in two different packages are clearly identified and can always be specifically designated. The C names are usually a good indication of the names of the future packages and should be stripped — it is possible to use the full name if useful. For example, here's how the following declaration and call could be translated:

[C]

Listing 2271: reg_interface.h

```
1 void registerInterface_Initialize (int size);
```

Listing 2272: reg_interface_test.c

```
1 #include "reg_interface.h"
2
3 int main(int argc, const char * argv[])
4 {
5     registerInterface_Initialize(15);
6
7     return 0;
8 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Namespaces
MD5: e8c25da648a2e8662d97a9a5b863a5bc

[Ada]

Listing 2273: register_interface.ads

```
1 package Register_Interface is
2     procedure Initialize (Size : Integer)
3         with Import      => True,
4             Convention => C,
5             External_Name => "registerInterface_Initialize";
6
7 end Register_Interface;
```

Listing 2274: main.adb

```
1 with Register_Interface;
2
3 procedure Main is
4 begin
5     Register_Interface.Initialize (15);
6 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Namespaces
MD5: 934edd7d3c74d058f862a786582a32c0

Note that in the above example, a **use** clause on Register_Interface could allow us to omit the prefix.

86.8 Pointers

The first thing to ask when translating pointers from C to Ada is: are they needed in the first place? In Ada, pointers (or access types) should only be used with complex structures that cannot be allocated at run-time — think of a linked list or a graph for example. There are many other situations that would need a pointer in C, but do not in Ada, in particular:

- Arrays, even when dynamically allocated
- Results of functions
- Passing large structures as parameters

- Access to registers
- ... others

This is not to say that pointers aren't used in these cases but, more often than not, the pointer is hidden from the user and automatically handled by the code generated by the compiler; thus avoiding possible mistakes from being made. Generally speaking, when looking at C code, it's good practice to start by analyzing how many pointers are used and to translate as many as possible into *pointerless* Ada structures.

Here are a few examples of such patterns — additional examples can be found throughout this document.

Dynamically allocated arrays can be directly allocated on the stack:

[C]

Listing 2275: array_decl.c

```

1  #include <stdlib.h>
2
3  int main() {
4      int *a = malloc(sizeof(int) * 10);
5
6      return 0;
7  }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_Stack_Alloc_C
MD5: a922c3e163494339d6773c6ab1256549

[Ada]

Listing 2276: main.adb

```

1  procedure Main is
2      type Arr is array (Integer range <>) of Integer;
3      A : Arr (0 .. 9);
4  begin
5      null;
6  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_Stack_Alloc_Ada
MD5: 2e4196c2a2016244a48de153beaa2b49

It's even possible to create a such an array within a structure, provided that the size of the array is known when instantiating this object, using a type discriminant:

[C]

Listing 2277: array_decl.c

```

1  #include <stdlib.h>
2
3  typedef struct {
4      int * a;
5  } S;
6
7  int main(int argc, const char * argv[])
8  {
9      S v;
```

(continues on next page)

(continued from previous page)

```
10
11     v.a = malloc(sizeof(int) * 10);
12
13     return 0;
14 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Struct_Array_Stack_Alloc_C
MD5: f8e5a877977387986b3e2353834a2989

[Ada]

Listing 2278: main.adb

```
1  procedure Main is
2      type Arr is array (Integer range <>) of Integer;
3
4      type S (Last : Integer) is record
5          A : Arr (0 .. Last);
6      end record;
7
8      V : S (9);
9  begin
10     null;
11 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Struct_Array_Stack_Alloc_Ada
MD5: 955c704bdb4b2b788e4a790ade12df7

With regards to parameter passing, usage mode (input / output) should be preferred to implementation mode (by copy or by reference). The Ada compiler will automatically pass a reference when needed. This works also for smaller objects, so that the compiler will copy in an out when needed. One of the advantages of this approach is that it clarifies the nature of the object: in particular, it differentiates between arrays and scalars. For example:

[C]

Listing 2279: p.h

```
1 void p (int * a, int * b);
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_In_Out_C
MD5: c2c936dd3afc4850c5869e4db73bb36b

[Ada]

Listing 2280: array_types.ads

```
1  package Array_Types is
2      type Arr is array (Integer range <>) of Integer;
3
4      procedure P (A : in out Integer; B : in out Arr);
5  end Array_Types;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Array_In_Out_Ada
 MD5: cf8e51391c9fd8608183c9dae2aa2802

Most of the time, access to registers end up in some specific structures being mapped onto a specific location in memory. In Ada, this can be achieved through an **Address** clause associated to a variable, for example:

[C]

Listing 2281: test_c.c

```
1 int main(int argc, const char * argv[])
2 {
3     int * r = (int *)0xFFFF00A0;
4
5     return 0;
6 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Address_C
 MD5: e810538d72d835a04736fcdf732f1930

[Ada]

Listing 2282: test.adb

```
1 with System;
2
3 procedure Test is
4     R : Integer with Address => System'To_Address (16#FFFF00A0#);
5 begin
6     null;
7 end Test;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Address_Ada
 MD5: 1263f7289cec6673f19d88bffbbee48

These are some of the most common misuse of pointers in Ada. Previous sections of the document deal with specifically using access types if absolutely necessary.

86.9 Bitwise Operations

Bitwise operations such as masks and shifts in Ada should be relatively rarely needed, and, when translating C code, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other.

Consider the case of using a C primitive type as a container for single bit boolean flags. In C, this would be done through masks, e.g.:

[C]

Listing 2283: flags.c

```
1 #define FLAG_1 0b0001
2 #define FLAG_2 0b0010
```

(continues on next page)

(continued from previous page)

```
3 #define FLAG_3 0b0100
4 #define FLAG_4 0b1000
5
6 int main(int argc, const char * argv[])
7 {
8     int value = 0;
9
10    value |= FLAG_2 | FLAG_4;
11
12    return 0;
13 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Flags_C
MD5: cf903dee1fb1d78d74dc42b66adcd5

In Ada, the above can be represented through a Boolean array of enumerate values:
[Ada]

Listing 2284: main.adb

```
1 procedure Main is
2     type Values is (Flag_1, Flag_2, Flag_3, Flag_4);
3     type Value_Array is array (Values) of Boolean
4         with Pack;
5
6     Value : Value_Array :=
7         (Flag_2 => True,
8          Flag_4 => True,
9          others => False);
10 begin
11     null;
12 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Flags_Ada
MD5: c92c8532763469f5e4d1027df2bd6a6b

Note the Pack directive for the array, which requests that the array takes as little space as possible.

It is also possible to map records on memory when additional control over the representation is needed or more complex data are used:

[C]

Listing 2285: struct_map.c

```
1 int main(int argc, const char * argv[])
2 {
3     int value = 0;
4
5     value = (2 << 1) | 1;
6
7     return 0;
8 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Rec_Map_C
 MD5: 16606f11ab3e9c86d3e1d88ac9c3f37f

[Ada]

Listing 2286: main.adb

```

1  procedure Main is
2      type Value_Rec is record
3          V1 : Boolean;
4          V2 : Integer range 0 .. 3;
5      end record;
6
7      for Value_Rec use record
8          V1 at 0 range 0 .. 0;
9          V2 at 0 range 1 .. 2;
10     end record;
11
12     Value : Value_Rec := (V1 => True, V2 => 2);
13 begin
14     null;
15 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Rec_Map_Ada
 MD5: 52078824814b0d83789dd837ac2e86bf

The benefit of using Ada structure instead of bitwise operations is threefold:

- The code is simpler to read / write and less error-prone
- Individual fields are named
- The compiler can run consistency checks (for example, check that the value indeed fit in the expected size).

Note that, in cases where bitwise operators are needed, Ada provides modular types with **and**, **or** and **xor** operators. Further shift operators can also be provided upon request through a **pragma**. So the above could also be literally translated to:

[Ada]

Listing 2287: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Value_Type is mod 2 ** 32;
5      pragma Provide_Shift_Operators (Value_Type);
6
7      Value : Value_Type;
8  begin
9      Value := Shift_Left (2, 1) or 1;
10     Put_Line ("Value = " & Value_Type'Image (Value));
11 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitwise_Ops_Ada
 MD5: 22cb824a0c99bd1a9092dc5f90e9d7fc

Runtime output


```
Value = 5
```

86.10 Mapping Structures to Bit-Fields

In the previous section, we've seen how to perform bitwise operations. In this section, we look at how to interpret a data type as a bit-field and perform low-level operations on it.

In general, you can create a bit-field from any arbitrary data type. First, we declare a bit-field type like this:

[Ada]

```
type Bit_Field is array (Natural range <>) of Boolean with Pack;
```

As we've seen previously, the `Pack` aspect declared at the end of the type declaration indicates that the compiler should optimize for size. We must use this aspect to be able to interpret data types as a bit-field.

Then, we can use the `Size` and the `Address` attributes of an object of any type to declare a bit-field for this object. We've discussed the `Size` attribute [earlier in this course](#) (page 1835).

The `Address` attribute indicates the address in memory of that object. For example, assuming we've declare a variable `V`, we can declare an actual bit-field object by referring to the `Address` attribute of `V` and using it in the declaration of the bit-field, as shown here:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address;
```

Note that, in this declaration, we're using the `Address` attribute of `V` for the `Address` aspect of `B`.

This technique is called overlays for serialization. Now, any operation that we perform on `B` will have a direct impact on `V`, since both are using the same memory location.

The approach that we use in this section relies on the `Address` aspect. Another approach would be to use unchecked conversions, which we'll discuss in the [next section](#) (page 1891).

We should add the `Volatile` aspect to the declaration to cover the case when both objects can still be changed independently — they need to be volatile, otherwise one change might be missed. This is the updated declaration:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address, Volatile;
```

Using the `Volatile` aspect is important at high level of optimizations. You can find further details about this aspect in the section about the [Volatile and Atomic aspects](#) (page 1830).

Another important aspect that should be added is `Import`. When used in the context of object declarations, it'll avoid default initialization which could overwrite the existing content while creating the overlay — see an example in the admonition below. The declaration now becomes:

```
B : Bit_Field (0 .. V'Size - 1)
  with
    Address => V'Address, Import, Volatile;
```

Let's look at a simple example:

[Ada]

Listing 2288: simple_bitfield.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Bitfield is
4   type Bit_Field is array (Natural range <>) of Boolean with Pack;
5
6   V : Integer := 0;
7   B : Bit_Field (0 .. V'Size - 1)
8     with Address => V'Address, Import, Volatile;
9 begin
10   B (2) := True;
11   Put_Line ("V = " & Integer'Image (V));
12 end Simple_Bitfield;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Ada
MD5: 193a2db91619426a145cd267f873145f

Runtime output

V = 4

In this example, we first initialize V with zero. Then, we use the bit-field B and set the third element (B (2)) to **True**. This automatically sets bit #3 of V to 1. Therefore, as expected, the application displays the message V = 4, which corresponds to $2^2 = 4$.

Note that, in the declaration of the bit-field type above, we could also have used a positive range. For example:

```

type Bit_Field is array (Positive range <>) of Boolean with Pack;

B : Bit_Field (1 .. V'Size)
  with Address => V'Address, Import, Volatile;

```

The only difference in this case is that the first bit is B (1) instead of B (0).

In C, we would rely on bit-shifting and masking to set that specific bit:

[C]

Listing 2289: bitfield.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5   int v = 0;
6
7   v = v | (1 << 2);
8
9   printf("v = %d\n", v);
10
11   return 0;
12 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_C
MD5: 98557f80ea3bc1b081ae2688f844cbe1

Runtime output

v = 4

i Important

Ada has the concept of default initialization. For example, you may set the default value of record components:

[Ada]

Listing 2290: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5     type Rec is record
6         X : Integer := 10;
7         Y : Integer := 11;
8     end record;
9
10    R : Rec;
11 begin
12    Put_Line ("R.X = " & Integer'Image (R.X));
13    Put_Line ("R.Y = " & Integer'Image (R.Y));
14 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Default_Record_Type
MD5: 010877f4d20302a1abcb9562c9e36a38

Runtime output

```

R.X = 10
R.Y = 11
```

In the code above, we don't explicitly initialize the components of R, so they still have the default values 10 and 11, which are displayed by the application.

Likewise, the `Default_Value` aspect can be used to specify the default value in other kinds of type declarations. For example:

[Ada]

Listing 2291: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5     type Percentage is range 0 .. 100
6         with Default_Value => 10;
7
8     P : Percentage;
9 begin
10    Put_Line ("P = " & Percentage'Image (P));
11 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Default_Value_Type
MD5: b3715f7cba0cbefa433bac529d95e395

Runtime output

```
P = 10
```

When declaring an object whose type has a default value, the object will automatically be initialized with the default value. In the example above, P is automatically initialized with 10, which is the default value of the Percentage type.

Some types have an implicit default value. For example, access types have a default value of **null**.

As we've just seen, when declaring objects for types with associated default values, automatic initialization will happen. This can also happen when creating an overlay with the **Address** aspect. The default value is then used to overwrite the content at the memory location indicated by the address. However, in most situations, this isn't the behavior we expect, since overlays are usually created to analyze and manipulate existing values. Let's look at an example where this happens:

[Ada]

Listing 2292: p.ads

```
1 package P is
2
3     type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;
4
5     type Byte_Field is array (Natural range <>) of Unsigned_8;
6
7     procedure Display_Bytes_Increment (V : in out Integer);
8 end P;
```

Listing 2293: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5     procedure Display_Bytes_Increment (V : in out Integer) is
6         BF : Byte_Field (1 .. V'Size / 8)
7             with Address => V'Address, Volatile;
8     begin
9         for B of BF loop
10             Put_Line ("Byte = " & Unsigned_8'Image (B));
11         end loop;
12         Put_Line ("Now incrementing...");
13         V := V + 1;
14     end Display_Bytes_Increment;
15
16 end P;
```

Listing 2294: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P; use P;
4
5 procedure Main is
6     V : Integer := 10;
7 begin
8     Put_Line ("V = " & Integer'Image (V));
9     Display_Bytes_Increment (V);
10    Put_Line ("V = " & Integer'Image (V));
11 end Main;
```

Code block metadata

```
Project: Courses.Ada_For_Embedded_C_Dev.Translation.Overlay_Default_Init_
↳Overwrite
MD5: 04994b2b4c98e9232a155515dc0c365a
```

Build output

```
p.adb:7:14: warning: default initialization of "Bf" may modify "V" [-gnatwo]
p.adb:7:14: warning: use pragma Import for "Bf" to suppress initialization (RM_
↳B.1(24)) [-gnatwo]
```

Runtime output

```
V = 10
Byte = 0
Byte = 0
Byte = 0
Byte = 0
Now incrementing...
V = 1
```

In this example, we expect `Display_Bytes_Increment` to display each byte of the `V` parameter and then increment it by one. Initially, `V` is set to 10, and the call to `Display_Bytes_Increment` should change it to 11. However, due to the default value associated to the `Unsigned_8` type — which is set to 0 — the value of `V` is overwritten in the declaration of `BF` (in `Display_Bytes_Increment`). Therefore, the value of `V` is 1 after the call to `Display_Bytes_Increment`. Of course, this is not the behavior that we originally intended.

Using the `Import` aspect solves this problem. This aspect tells the compiler to not apply default initialization in the declaration because the object is imported. Let's look at the corrected example:

[Ada]

Listing 2295: p.ads

```
1 package P is
2
3     type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;
4
5     type Byte_Field is array (Natural range <>) of Unsigned_8;
6
7     procedure Display_Bytes_Increment (V : in out Integer);
8 end P;
```

Listing 2296: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5     procedure Display_Bytes_Increment (V : in out Integer) is
6         BF : Byte_Field (1 .. V'Size / 8)
7             with Address => V'Address, Import, Volatile;
8     begin
9         for B of BF loop
10             Put_Line ("Byte = " & Unsigned_8'Image (B));
11         end loop;
12         Put_Line ("Now incrementing...");
13         V := V + 1;
14     end Display_Bytes_Increment;
15
16 end P;
```

Listing 2297: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P; use P;
4
5 procedure Main is
6   V : Integer := 10;
7 begin
8   Put_Line ("V = " & Integer'Image (V));
9   Display_Bytes_Increment (V);
10  Put_Line ("V = " & Integer'Image (V));
11 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Overlay_Default_Init_Import
MD5: e269d9d3c06c0f6c69ead16e7d2ba70b

Runtime output

```

V = 10
Byte = 10
Byte = 0
Byte = 0
Byte = 0
Now incrementing...
V = 11

```

This unwanted side-effect of the initialization by the `Default_Value` aspect that we've just seen can also happen in these cases:

- when we set a default value for components of a record type declaration,
- when we use the `Default_Component_Value` aspect for array types, or
- when we set use the `Initialize Scalars` pragma for a package.

Again, using the `Import` aspect when declaring the overlay eliminates this side-effect.

We can use this pattern for objects of more complex data types like arrays or records. For example:

[Ada]

Listing 2298: int_array_bitfield.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Int_Array_Bitfield is
4   type Bit_Field is array (Natural range <>) of Boolean with Pack;
5
6   A : array (1 .. 2) of Integer := (others => 0);
7   B : Bit_Field (0 .. A'Size - 1)
8     with Address => A'Address, Import, Volatile;
9 begin
10  B (2) := True;
11  for I in A'Range loop
12    Put_Line ("A (" & Integer'Image (I)
13              & ") = " & Integer'Image (A (I)));
14  end loop;
15 end Int_Array_Bitfield;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Int_Array_Ada
MD5: 478ba4ce4f5886566556bddb58245eb9

Runtime output

```
A ( 1)= 4  
A ( 2)= 0
```

In the Ada example above, we're using the bit-field to set bit #3 of the first element of the array (A (1)). We could set bit #4 of the second element by using the size of the data type (in this case, `Integer'Size`):

[Ada]

```
B (Integer'Size + 3) := True;
```

In C, we would select the specific array position and, again, rely on bit-shifting and masking to set that specific bit:

[C]

Listing 2299: bitfield_int_array.c

```
1  #include <stdio.h>  
2  
3  int main(int argc, const char * argv[])  
4  {  
5      int i;  
6      int a[2] = {0, 0};  
7  
8      a[0] = a[0] | (1 << 2);  
9  
10     for (i = 0; i < 2; i++)  
11     {  
12         printf("a[%d] = %d\n", i, a[i]);  
13     }  
14  
15     return 0;  
16 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Int_Array_C
MD5: 4dc3fe77e8260ff3b449c8779745a63c

Runtime output

```
a[0] = 4  
a[1] = 0
```

Since we can use this pattern for any arbitrary data type, this allows us to easily create a subprogram to serialize data types and, for example, transmit complex data structures as a bitstream. For example:

[Ada]

Listing 2300: serializer.ads

```
1  package Serializer is  
2  
3      type Bit_Field is array (Natural range <>) of Boolean with Pack;  
4
```

(continues on next page)

(continued from previous page)

```

5  procedure Transmit (B : Bit_Field);
6
7  end Serializer;

```

Listing 2301: serializer.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Serializer is
4
5      procedure Transmit (B : Bit_Field) is
6
7          procedure Show_Bit (V : Boolean) is
8              begin
9                  case V is
10                     when False => Put ("0");
11                     when True  => Put ("1");
12                 end case;
13             end Show_Bit;
14
15         begin
16             Put ("Bits: ");
17             for E of B loop
18                 Show_Bit (E);
19             end loop;
20             New_Line;
21         end Transmit;
22
23 end Serializer;

```

Listing 2302: my_recs.ads

```

1  package My_Recs is
2
3      type Rec is record
4          V : Integer;
5          S : String (1 .. 3);
6      end record;
7
8  end My_Recs;

```

Listing 2303: main.adb

```

1  with Serializer; use Serializer;
2  with My_Recs;   use My_Recs;
3
4  procedure Main is
5      R : Rec := (5, "abc");
6      B : Bit_Field (0 .. R'Size - 1)
7          with Address => R'Address, Import, Volatile;
8  begin
9      Transmit (B);
10 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Serialization_ada
MD5: 5c9c2d18bab7c78456d1d795c6334cd9

Build output


```
main.adb:9:14: warning: volatile actual passed by copy (RM C.6(19)) [enabled by _  
↳default]
```

Runtime output

```
Bits: 1010000000000000000000000000000010000110010001101100011000000000
```

In this example, the Transmit procedure from Serializer package displays the individual bits of a bit-field. We could have used this strategy to actually transmit the information as a bitstream. In the main application, we call Transmit for the object R of record type Rec. Since Transmit has the bit-field type as a parameter, we can use it for any type, as long as we have a corresponding bit-field representation.

In C, we interpret the input pointer as an array of bytes, and then use shifting and masking to access the bits of that byte. Here, we use the **char** type because it has a size of one byte in most platforms.

[C]

Listing 2304: my_recs.h

```
1 typedef struct {  
2     int v;  
3     char s[4];  
4 } rec;
```

Listing 2305: serializer.h

```
1 void transmit (void *bits, int len);
```

Listing 2306: serializer.c

```
1 #include "serializer.h"  
2  
3 #include <stdio.h>  
4 #include <assert.h>  
5  
6 void transmit (void *bits, int len)  
7 {  
8     int i, j;  
9     char *c = (char *)bits;  
10  
11     assert(sizeof(char) == 1);  
12  
13     printf("Bits: ");  
14     for (i = 0; i < len / (sizeof(char) * 8); i++)  
15     {  
16         for (j = 0; j < sizeof(char) * 8; j++)  
17         {  
18             printf("%d", c[i] >> j & 1);  
19         }  
20     }  
21     printf("\n");  
22 }
```

Listing 2307: bitfield_serialization.c

```
1 #include <stdio.h>  
2  
3 #include "my_recs.h"  
4 #include "serializer.h"
```

(continues on next page)

(continued from previous page)

```

5
6 int main(int argc, const char * argv[])
7 {
8     rec r = {5, "abc"};
9
10    transmit(&r, sizeof(r) * 8);
11
12    return 0;
13 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Serialization_C
MD5: 47f0a4efcbec9303f44d535064e5d6ce

Runtime output

Bits: 1010000000000000000000000000000010000110010001101100011000000000

Similarly, we can write a subprogram that converts a bit-field — which may have been received as a bitstream — to a specific type. We can add a `To_Rec` subprogram to the `My_Recs` package to convert a bit-field to the `Rec` type. This can be used to convert a bitstream that we received into the actual data type representation.

As you know, we may write the `To_Rec` subprogram as a procedure or as a function. Since we need to use slightly different strategies for the implementation, the following example has both versions of `To_Rec`.

This is the updated code for the `My_Recs` package and the `Main` procedure:

[Ada]

Listing 2308: serializer.ads

```

1 package Serializer is
2
3     type Bit_Field is array (Natural range <>) of Boolean with Pack;
4
5     procedure Transmit (B : Bit_Field);
6
7 end Serializer;

```

Listing 2309: serializer.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Serializer is
4
5     procedure Transmit (B : Bit_Field) is
6
7         procedure Show_Bit (V : Boolean) is
8         begin
9             case V is
10                 when False => Put ("0");
11                 when True  => Put ("1");
12             end case;
13         end Show_Bit;
14
15     begin
16         Put ("Bits: ");
17         for E of B loop

```

(continues on next page)

(continued from previous page)

```
18     Show_Bit (E);
19     end loop;
20     New_Line;
21 end Transmit;
22
23 end Serializer;
```

Listing 2310: my_recs.ads

```
1 with Serializer; use Serializer;
2
3 package My_Recs is
4
5     type Rec is record
6         V : Integer;
7         S : String (1 .. 3);
8     end record;
9
10    procedure To_Rec (B : Bit_Field;
11                    R : out Rec);
12
13    function To_Rec (B : Bit_Field) return Rec;
14
15    procedure Display (R : Rec);
16
17 end My_Recs;
```

Listing 2311: my_recs.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Recs is
4
5     procedure To_Rec (B : Bit_Field;
6                     R : out Rec) is
7         B_R : Rec
8         with Address => B'Address, Import, Volatile;
9     begin
10         -- Assigning data from overlayed record B_R to output parameter R.
11         R := B_R;
12     end To_Rec;
13
14    function To_Rec (B : Bit_Field) return Rec is
15        R : Rec;
16        B_R : Rec
17        with Address => B'Address, Import, Volatile;
18    begin
19        -- Assigning data from overlayed record B_R to local record R.
20        R := B_R;
21
22        return R;
23    end To_Rec;
24
25    procedure Display (R : Rec) is
26    begin
27        Put "(" & Integer'Image (R.V) & ", "
28            & (R.S) & ")");
29    end Display;
30
31 end My_Recs;
```

Listing 2312: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Serializer; use Serializer;
3 with My_Recs; use My_Recs;
4
5 procedure Main is
6   R1 : Rec := (5, "abc");
7   R2 : Rec := (0, "zzz");
8
9   B1 : Bit_Field (0 .. R1'Size - 1)
10    with Address => R1'Address, Import, Volatile;
11 begin
12   Put ("R2 = ");
13   Display (R2);
14   New_Line;
15
16   -- Getting Rec type using data from B1, which is a bit-field
17   -- representation of R1.
18   To_Rec (B1, R2);
19
20   -- We could use the function version of To_Rec:
21   -- R2 := To_Rec (B1);
22
23   Put_Line ("New bitstream received!");
24   Put ("R2 = ");
25   Display (R2);
26   New_Line;
27 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Deserialization_Ada
MD5: bf5cb5ef048ed1f95dba8e85275f6e32

Build output

```
main.adb:18:12: warning: volatile actual passed by copy (RM C.6(19)) [enabled by default]
```

Runtime output

```

R2 = ( 0, zzz)
New bitstream received!
R2 = ( 5, abc)

```

In both versions of `To_Rec`, we declare the record object `B_R` as an overlay of the input bit-field. In the procedure version of `To_Rec`, we then simply copy the data from `B_R` to the output parameter `R`. In the function version of `To_Rec`, however, we need to declare a local record object `R`, which we return after the assignment.

In C, we can interpret the input pointer as an array of bytes, and copy the individual bytes. For example:

```
[C]
```

Listing 2313: my_recs.h

```

1 typedef struct {
2     int v;
3     char s[3];
4 } rec;

```

(continues on next page)

(continued from previous page)

```
5
6 void to_r (void *bits, int len, rec *r);
7
8 void display_r (rec *r);
```

Listing 2314: my_recs.c

```
1 #include "my_recs.h"
2
3 #include <stdio.h>
4 #include <assert.h>
5
6 void to_r (void *bits, int len, rec *r)
7 {
8     int i;
9     char *c1 = (char *)bits;
10    char *c2 = (char *)r;
11
12    assert(len == sizeof(rec) * 8);
13
14    for (i = 0; i < len / (sizeof(char) * 8); i++)
15    {
16        c2[i] = c1[i];
17    }
18 }
19
20 void display_r (rec *r)
21 {
22     printf("{%d, %c%c%c}", r->v, r->s[0], r->s[1], r->s[2]);
23 }
```

Listing 2315: bitfield_serialization.c

```
1 #include <stdio.h>
2 #include "my_recs.h"
3
4 int main(int argc, const char * argv[])
5 {
6     rec r1 = {5, "abc"};
7     rec r2 = {0, "zzz"};
8
9     printf("r2 = ");
10    display_r (&r2);
11    printf("\n");
12
13    to_r(&r1, sizeof(r1) * 8, &r2);
14
15    printf("New bitstream received!\n");
16    printf("r2 = ");
17    display_r (&r2);
18    printf("\n");
19
20    return 0;
21 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Deserialization_C
MD5: 1c0fda773b0b681d0a4e9a57cf67d997

Runtime output

```
r2 = {0, zzz}
New bitstream received!
r2 = {5, abc}
```

Here, `to_r` casts both pointer parameters to pointers to **char** to get a byte-aligned pointer. Then, it simply copies the data byte-by-byte.

86.10.1 Overlays vs. Unchecked Conversions

Unchecked conversions are another way of converting between unrelated data types. This conversion is done by instantiating the generic `Unchecked_Conversions` function for the types you want to convert. Let's look at a simple example:

[Ada]

Listing 2316: `simple_unchecked_conversion.adb`

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Unchecked_Conversion;
3
4 procedure Simple_Unchecked_Conversion is
5   type State is (Off, State_1, State_2)
6     with Size => Integer'Size;
7
8   for State use (Off => 0, State_1 => 32, State_2 => 64);
9
10  function As_Integer is new Ada.Unchecked_Conversion (Source => State,
11                                                         Target => Integer);
12
13  I : Integer;
14 begin
15    I := As_Integer (State_2);
16    Put_Line ("I = " & Integer'Image (I));
17 end Simple_Unchecked_Conversion;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Simple_Unchecked_Conversion
MD5: 1b6058ef1919879a7d2d86be41f3b269

Runtime output

```
I = 64
```

In this example, `As_Integer` is an instantiation of `Unchecked_Conversion` to convert between the `State` enumeration and the **Integer** type. Note that, in order to ensure safe conversion, we're declaring `State` to have the same size as the **Integer** type we want to convert to.

This is the corresponding implementation using overlays:

[Ada]

Listing 2317: `simple_overlay.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Overlay is
4   type State is (Off, State_1, State_2)
5     with Size => Integer'Size;
6
7   for State use (Off => 0, State_1 => 32, State_2 => 64);
```

(continues on next page)

(continued from previous page)

```

8
9   S : State;
10  I : Integer
11    with Address => S'Address, Import, Volatile;
12 begin
13   S := State_2;
14   Put_Line ("I = " & Integer'Image (I));
15 end Simple_Overlay;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Simple_Overlay
MD5: 932135a47c36c406e70b22e075afeaf2

Runtime output

```
I = 64
```

Let's look at another example of converting between different numeric formats. In this case, we want to convert between a 16-bit fixed-point and a 16-bit integer data type. This is how we can do it using `Unchecked_Conversion`:

[Ada]

Listing 2318: fixed_int_unchecked_conversion.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Unchecked_Conversion;
3
4 procedure Fixed_Int_Unchecked_Conversion is
5   Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
6   Max_16   : constant := 2 ** 15;
7
8   type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
9     with Size => 16;
10  type Int_16   is range -Max_16 .. Max_16 - 1
11    with Size => 16;
12
13  function As_Int_16 is new Ada.Unchecked_Conversion (Source => Fixed_16,
14                                                       Target => Int_16);
15  function As_Fixed_16 is new Ada.Unchecked_Conversion (Source => Int_16,
16                                                         Target => Fixed_16);
17
18  I : Int_16   := 0;
19  F : Fixed_16 := 0.0;
20 begin
21  F := Fixed_16'Last;
22  I := As_Int_16 (F);
23
24  Put_Line ("F = " & Fixed_16'Image (F));
25  Put_Line ("I = " & Int_16'Image (I));
26 end Fixed_Int_Unchecked_Conversion;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Fixed_Int_Unchecked_Conversion
MD5: 53b59ca56a5c25408d8b6e5fcb06f37a

Runtime output

```
F = 0.99997
I = 32767
```

Here, we instantiate `Unchecked_Conversion` for the `Int_16` and `Fixed_16` types, and we call the instantiated functions explicitly. In this case, we call `As_Int_16` to get the integer value corresponding to `Fixed_16'Last`.

This is how we can rewrite the implementation above using overlays:

[Ada]

Listing 2319: `fixed_int_overlay.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Fixed_Int_Overlay is
4      Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
5      Max_16   : constant := 2 ** 15;
6
7      type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
8          with Size => 16;
9      type Int_16   is range -Max_16 .. Max_16 - 1
10         with Size => 16;
11
12     I : Int_16 := 0;
13     F : Fixed_16
14         with Address => I'Address, Import, Volatile;
15 begin
16     F := Fixed_16'Last;
17
18     Put_Line ("F = " & Fixed_16'Image (F));
19     Put_Line ("I = " & Int_16'Image (I));
20 end Fixed_Int_Overlay;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Translation.Fixed_Int_Overlay
MD5: ee86e3d10266f8c8c96311595b6624ec

Runtime output

```

F = 0.99997
I = 32767
```

Here, the conversion to the integer value is implicit, so we don't need to call a conversion function.

Using `Unchecked_Conversion` has the advantage of making it clear that a conversion is happening, since the conversion is written explicitly in the code. With overlays, that conversion is automatic and therefore implicit. In that sense, using an unchecked conversion is a cleaner and safer approach. On the other hand, an unchecked conversion requires a copy, so it's less efficient than overlays, where no copy is performed — because one change in the source object is automatically reflected in the target object (and vice-versa). In the end, the choice between unchecked conversions and overlays depends on the level of performance that you want to achieve.

Also note that an unchecked conversion only has defined behavior when instantiated for constrained types. For example, we shouldn't use this kind of conversion:

```

Ada.Unchecked_Conversion (Source => String,
                          Target => Integer);
```

Although this compiles, the behavior will only be well-defined in those cases when `Source'Size = Target'Size`. Therefore, instead of using an unconstrained type for `Source`, we should use a subtype that matches this expectation:


```
subtype Integer_String is String (1 .. Integer'Size / Character'Size);

function As_Integer is new
  Ada.Unchecked_Conversion (Source => Integer_String,
                           Target => Integer);
```

Similarly, in order to rewrite the examples using bit-fields that we've seen in the previous section, we cannot simply instantiate `Unchecked_Conversion` with the `Target` indicating the *unconstrained* bit-field, such as:

```
Ada.Unchecked_Conversion (Source => Integer,
                          Target => Bit_Field);
```

Instead, we have to declare a subtype for the specific range we're interested in. This is how we can rewrite one of the previous examples:

[Ada]

Listing 2320: simple_bitfield_conversion.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Unchecked_Conversion;
3
4 procedure Simple_Bitfield_Conversion is
5   type Bit_Field is array (Natural range <>) of Boolean with Pack;
6
7   V : Integer := 4;
8
9   -- Declaring subtype that takes the size of V into account.
10  --
11  subtype Integer_Bit_Field is Bit_Field (0 .. V'Size - 1);
12
13  -- NOTE: we could also use the Integer type in the declaration:
14  --
15  --   subtype Integer_Bit_Field is Bit_Field (0 .. Integer'Size - 1);
16  --
17
18  -- Using the Integer_Bit_Field subtype as the target
19  function As_Bit_Field is new
20    Ada.Unchecked_Conversion (Source => Integer,
21                             Target => Integer_Bit_Field);
22
23  B : Integer_Bit_Field;
24 begin
25  B := As_Bit_Field (V);
26
27  Put_Line ("V = " & Integer'Image (V));
28 end Simple_Bitfield_Conversion;
```

Code block metadata

```
Project: Courses.Ada_For_Embedded_C_Dev.Translation.Bitfield_Conversion
MD5: 46ead7e5f3da8f261770811d450453e7
```

Runtime output

```
V = 4
```

In this example, we first declare the subtype `Integer_Bit_Field` as a bit-field with a length that fits the `V` variable we want to convert to. Then, we can use that subtype in the instantiation of `Unchecked_Conversion`.

HANDLING VARIABILITY AND RE-USABILITY

87.1 Understanding static and dynamic variability

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. For example, the same application may need to be portable between two different architectures (ARM and x86), or two different platforms with different set of devices available. Maybe the same application is used for two different generations of the product, so it needs to account for absence or presence of new features, or it's used for different projects which may select different components or configurations. All these cases, and many others, require variability in the software in order to ensure its reusability.

In C, variability is usually achieved through macros and function pointers, the former being tied to static variability (variability in different builds) the latter to dynamic variability (variability within the same build decided at run-time).

Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. When Ada isn't enough, the GNAT compilation system also provides a layer of capabilities, in particular selection of alternate bodies.

If you're familiar with object-oriented programming (OOP) — supported in languages such as C++ and Java —, you might also be interested in knowing that OOP is supported by Ada and can be used to implement variability. This should, however, be used with care, as OOP brings its own set of problems, such as loss of efficiency — dispatching calls can't be inlined and require one level of indirection — or loss of analyzability — the target of a dispatching call isn't known at run time. As a rule of thumb, OOP should be considered only for cases of dynamic variability, where several versions of the same object need to exist concurrently in the same application.

87.2 Handling variability & reusability statically

87.2.1 Genericity

One usage of C macros involves the creation of functions that works regardless of the type they're being called upon. For example, a swap macro may look like:

[C]

Listing 2321: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define SWAP(t, a, b) ({\
5     t tmp = a; \
6     a = b; \
```

(continues on next page)

(continued from previous page)

```
7         b = tmp; \
8     })
9
10  int main()
11  {
12      int a = 10;
13      int b = 42;
14
15      printf("a = %d, b = %d\n", a, b);
16
17      SWAP (int, a, b);
18
19      printf("a = %d, b = %d\n", a, b);
20
21      return 0;
22  }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Swap_C
MD5: 96d0e8ce9ae985e4de9ed64a0f0961f5

Runtime output

```
a = 10, b = 42
a = 42, b = 10
```

Ada offers a way to declare this kind of functions as a generic, that is, a function that is written after static arguments, such as a parameter:

[Ada]

Listing 2322: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      generic
6          type A_Type is private;
7          procedure Swap (Left, Right : in out A_Type);
8
9          procedure Swap (Left, Right : in out A_Type) is
10              Temp : constant A_Type := Left;
11          begin
12              Left := Right;
13              Right := Temp;
14          end Swap;
15
16          procedure Swap_I is new Swap (Integer);
17
18          A : Integer := 10;
19          B : Integer := 42;
20
21      begin
22          Put_Line ("A = "
23                  & Integer'Image (A)
24                  & ", B = "
25                  & Integer'Image (B));
26
27          Swap_I (A, B);
```

(continues on next page)

(continued from previous page)

```

28
29     Put_Line ("A = "
30               & Integer'Image (A)
31               & ", B = "
32               & Integer'Image (B));
33 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Swap_Ada
MD5: 13f3527b4e3258ebd43be827ad0fcd14

Runtime output

```

A = 10, B = 42
A = 42, B = 10
```

There are a few key differences between the C and the Ada version here. In C, the macro can be used directly and essentially get expanded by the preprocessor without any kind of checks. In Ada, the generic will first be checked for internal consistency. It then needs to be explicitly instantiated for a concrete type. From there, it's exactly as if there was an actual version of this Swap function, which is going to be called as any other function. All rules for parameter modes and control will apply to this instance.

In many respects, an Ada generic is a way to provide a safe specification and implementation of such macros, through both the validation of the generic itself and its usage.

Subprograms aren't the only entities that can be made generic. As a matter of fact, it's much more common to render an entire package generic. In this case the instantiation creates a new version of all the entities present in the generic, including global variables. For example:

[Ada]

Listing 2323: gen.ads

```

1 generic
2   type T is private;
3 package Gen is
4   type C is tagged record
5     V : T;
6   end record;
7
8   G : Integer;
9 end Gen;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Gen_Pkg_1
MD5: 721f9954561b7e0d2964ba0d226c748b

The above can be instantiated and used the following way:

Listing 2324: main.adb

```

1 with Gen;
2
3 procedure Main is
4   package I1 is new Gen (Integer);
5   package I2 is new Gen (Integer);
6   subtype Str10 is String (1 .. 10);
```

(continues on next page)

(continued from previous page)

```

7   package I3 is new Gen (Str10);
8   begin
9     I1.G := 0;
10    I2.G := 1;
11    I3.G := 2;
12  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Gen_Pkg_1
MD5: ab0e99dedf40fff1bced048a96a0fbb6

Here, I1.G, I2.G and I3.G are three distinct variables.

So far, we've only looked at generics with one kind of parameter: a so-called private type. There's actually much more that can be described in this section, such as variables, subprograms or package instantiations with certain properties. For example, the following provides a sort algorithm for any kind of structurally compatible array type:

[Ada]

Listing 2325: sort.ads

```

1  generic
2    type Component is private;
3    type Index is (<);
4    with function "<" (Left, Right : Component) return Boolean;
5    type Array_Type is array (Index range <>) of Component;
6  procedure Sort (A : in out Array_Type);

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Gen_Pkg_2
MD5: 5781f53f4fd4453ecc1313d05ab76f81

The declaration above states that we need a type (Component), a discrete type (Index), a comparison subprogram (" $<$ "), and an array definition (Array_Type). Given these, it's possible to write an algorithm that can sort any Array_Type. Note the usage of the with reserved word in front of the function name: it exists to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```

type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type e.g. String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type to T2

```

For a more complete list please reference the Generic Formal Types in the [Appendix of the Introduction to Ada course](#) (page 271).

87.2.2 Simple derivation

Let's take a case where a codebase needs to handle small variations of a given device, or maybe different generations of a device, depending on the platform it's running on. In this example, we're assuming that each platform will lead to a different binary, so the code can statically resolve which set of services are available. However, we want an easy way to

implement a new device based on a previous one, saying "this new device is the same as this previous device, with these new services and these changes in existing services".

We can implement such patterns using Ada's simple derivation — as opposed to tagged derivation, which is OOP-related and discussed in a later section.

Let's start from the following example:

[Ada]

Listing 2326: drivers_1.ads

```

1 package Drivers_1 is
2
3     type Device_1 is null record;
4     procedure Startup (Device : Device_1);
5     procedure Send (Device : Device_1; Data : Integer);
6     procedure Send_Fast (Device : Device_1; Data : Integer);
7     procedure Receive (Device : Device_1; Data : out Integer);
8
9 end Drivers_1;
```

Listing 2327: drivers_1.adb

```

1 package body Drivers_1 is
2
3     -- NOTE: unimplemented procedures: Startup, Send, Send_Fast
4     --       mock-up implementation: Receive
5
6     procedure Startup (Device : Device_1) is null;
7
8     procedure Send (Device : Device_1; Data : Integer) is null;
9
10    procedure Send_Fast (Device : Device_1; Data : Integer) is null;
11
12    procedure Receive (Device : Device_1; Data : out Integer) is
13    begin
14        Data := 42;
15    end Receive;
16
17 end Drivers_1;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 4f9d7e29b64cda8664438a1d7eed9049

In the above example, Device_1 is an empty record type. It may also have some fields if required, or be a different type such as a scalar. Then the four procedures Startup, Send, Send_Fast and Receive are primitives of this type. A primitive is essentially a subprogram that has a parameter or return type directly referencing this type and declared in the same scope. At this stage, there's nothing special with this type: we're using it as we would use any other type. For example:

[Ada]

Listing 2328: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers_1;   use Drivers_1;
3
4 procedure Main is
5     D : Device_1;
```

(continues on next page)

(continued from previous page)

```
6   I : Integer;  
7   begin  
8     Startup (D);  
9     Send_Fast (D, 999);  
10    Receive (D, I);  
11    Put_Line (Integer'Image (I));  
12  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 1b28f2c8ca92498cbcd582f092b9912

Runtime output

42

Let's now assume that we need to implement a new generation of device, Device_2. This new device works exactly like the first one, except for the startup code that has to be done differently. We can create a new type that operates exactly like the previous one, but modifies only the behavior of Startup:

[Ada]

Listing 2329: drivers_2.ads

```
1  with Drivers_1; use Drivers_1;  
2  
3  package Drivers_2 is  
4  
5     type Device_2 is new Device_1;  
6  
7     overriding  
8     procedure Startup (Device : Device_2);  
9  
10  end Drivers_2;
```

Listing 2330: drivers_2.adb

```
1  package body Drivers_2 is  
2  
3     overriding  
4     procedure Startup (Device : Device_2) is null;  
5  
6  end Drivers_2;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 276c9da0b7c9ad61d679531e16fdd9cb

Here, Device_2 is derived from Device_1. It contains all the exact same properties and primitives, in particular, Startup, Send, Send_Fast and Receive. However, here, we decided to change the Startup function and to provide a different implementation. We override this function. The main subprogram doesn't change much, except for the fact that it now relies on a different type:

[Ada]

Listing 2331: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers_2; use Drivers_2;
3
4 procedure Main is
5     D : Device_2;
6     I : Integer;
7 begin
8     Startup (D);
9     Send_Fast (D, 999);
10    Receive (D, I);
11    Put_Line (Integer'Image (I));
12 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 31e7105a99771ce6c1602af117e2e8a6

Runtime output

42

We can continue with this approach and introduce a new generation of devices. This new device doesn't implement the `Send_Fast` service so we want to remove it from the list of available services. Furthermore, for the purpose of our example, let's assume that the hardware team went back to the `Device_1` way of implementing `Startup`. We can write this new device the following way:

[Ada]

Listing 2332: drivers_3.ads

```

1 with Drivers_1; use Drivers_1;
2
3 package Drivers_3 is
4
5     type Device_3 is new Device_1;
6
7     overriding
8     procedure Startup (Device : Device_3);
9
10    procedure Send_Fast (Device : Device_3; Data : Integer)
11    is abstract;
12
13 end Drivers_3;

```

Listing 2333: drivers_3.adb

```

1 package body Drivers_3 is
2
3     overriding
4     procedure Startup (Device : Device_3) is null;
5
6 end Drivers_3;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 779579532c81b672d8a641c0b8594ed5

The **is abstract** definition makes illegal any call to a function, so calls to `Send_Fast` on `Device_3` will be flagged as being illegal. To then implement `Startup` of `Device_3` as being the same as the `Startup` of `Device_1`, we can convert the type in the implementation:

[Ada]

Listing 2334: drivers_3.adb

```
1 package body Drivers_3 is
2
3     overriding
4     procedure Startup (Device : Device_3) is
5     begin
6         Drivers_1.Startup (Device_1 (Device));
7     end Startup;
8
9 end Drivers_3;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 5db9596c276a7a4521914f4108f61d28

Our Main now looks like:

[Ada]

Listing 2335: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers_3;   use Drivers_3;
3
4 procedure Main is
5     D : Device_3;
6     I : Integer;
7 begin
8     Startup (D);
9     Send_Fast (D, 999);
10    Receive (D, I);
11    Put_Line (Integer'Image (I));
12 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: 8b6af16d21c2f8a1f0e4866e6ddffdf1f

Build output

```
main.adb:9:04: error: cannot call abstract operation "Send_Fast" declared at ↵
↳ drivers_3.ads:10
gprbuild: *** compilation phase failed
```

Here, the call to `Send_Fast` will get flagged by the compiler.

Note that the fact that the code of `Main` has to be changed for every implementation isn't necessarily satisfactory. We may want to go one step further, and isolate the selection of the device kind to be used for the whole application in one unique file. One way to do this is to use the same name for all types, and use a renaming to select which package to use. Here's a simplified example to illustrate that:

[Ada]

Listing 2336: drivers_1.ads

```

1 package Drivers_1 is
2
3     type Transceiver is null record;
4     procedure Send (Device : Transceiver; Data : Integer);
5     procedure Receive (Device : Transceiver; Data : out Integer);
6
7 end Drivers_1;
```

Listing 2337: drivers_1.adb

```

1 package body Drivers_1 is
2
3     procedure Send (Device : Transceiver; Data : Integer) is null;
4
5     procedure Receive (Device : Transceiver; Data : out Integer) is
6         pragma Unreferenced (Device);
7     begin
8         Data := 42;
9     end Receive;
10
11 end Drivers_1;
```

Listing 2338: drivers_2.ads

```

1 with Drivers_1;
2
3 package Drivers_2 is
4
5     type Transceiver is new Drivers_1.Transceiver;
6     procedure Send (Device : Transceiver; Data : Integer);
7     procedure Receive (Device : Transceiver; Data : out Integer);
8
9 end Drivers_2;
```

Listing 2339: drivers_2.adb

```

1 package body Drivers_2 is
2
3     procedure Send (Device : Transceiver; Data : Integer) is null;
4
5     procedure Receive (Device : Transceiver; Data : out Integer) is
6         pragma Unreferenced (Device);
7     begin
8         Data := 42;
9     end Receive;
10
11 end Drivers_2;
```

Listing 2340: drivers.ads

```

1 with Drivers_1;
2
3 package Drivers renames Drivers_1;
```

Listing 2341: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers;      use Drivers;
```

(continues on next page)

(continued from previous page)

```

3
4 procedure Main is
5   D : Transceiver;
6   I : Integer;
7 begin
8   Send (D, 999);
9   Receive (D, I);
10  Put_Line (Integer'Image (I));
11 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Derived_Drivers
MD5: e92590e4b91fef33f4fec23362a52873

Runtime output

42

In the above example, the whole code can rely on `drivers.ads`, instead of relying on the specific driver. Here, `Drivers` is another name for `Driver_1`. In order to switch to `Driver_2`, the project only has to replace that one `drivers.ads` file.

In the following section, we'll go one step further and demonstrate that this selection can be done through a configuration switch selected at build time instead of a manual code modification.

87.2.3 Configuration pragma files

Configuration pragmas are a set of pragmas that modify the compilation of source-code files. You may use them to either relax or strengthen requirements. For example:

```
pragma Suppress (Overflow_Check);
```

In this example, we're suppressing the overflow check, thereby relaxing a requirement. Normally, the following program would raise a constraint error due to a failed overflow check:

[Ada]

Listing 2342: p.ads

```

1 package P is
2   function Add_Max (A : Integer) return Integer;
3 end P;
```

Listing 2343: p.adb

```

1 package body P is
2   function Add_Max (A : Integer) return Integer is
3   begin
4     return A + Integer'Last;
5   end Add_Max;
6 end P;
```

Listing 2344: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with P;           use P;
3
```

(continues on next page)

(continued from previous page)

```

4 procedure Main is
5   I : Integer := Integer'Last;
6 begin
7   I := Add_Max (I);
8   Put_Line ("I = " & Integer'Image (I));
9 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Constraint_Error_Detection
 MD5: d6960fe8ae2af1d66b617bb92d3d47b6

Runtime output

```
raised CONSTRAINT_ERROR : p.adb:4 overflow check failed
```

When suppressing the overflow check, however, the program doesn't raise an exception, and the value that `Add_Max` returns is -2, which is a wraparound of the sum of the maximum integer values (`Integer'Last + Integer'Last`).

We could also strengthen requirements, as in this example:

```
pragma Restrictions (No_Floating_Point);
```

Here, the restriction forbids the use of floating-point types and objects. The following program would violate this restriction, so the compiler isn't able to compile the program when the restriction is used:

```

procedure Main is
  F : Float := 0.0;
  -- Declaration is not possible with No_Floating_Point restriction.
begin
  null;
end Main;

```

Restrictions are especially useful for high-integrity applications. In fact, the Ada Reference Manual has [a separate section for them](http://www.ada-auth.org/standards/12rm/html/RM-H-4.html)⁴⁴⁷.

When creating a project, it is practical to list all configuration pragmas in a separate file. This is called a configuration pragma file, and it usually has an `.adc` file extension. If you use **GPRbuild** for building Ada applications, you can specify the configuration pragma file in the corresponding project file. For example, here we indicate that `gnat.adc` is the configuration pragma file for our project:

```

project Default is

  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Main use ("main.adb");

  package Compiler is
    for Local_Configuration_Pragmas use "gnat.adc";
  end Compiler;

end Default;

```

⁴⁴⁷ <http://www.ada-auth.org/standards/12rm/html/RM-H-4.html>

87.2.4 Configuration packages

In C, preprocessing flags are used to create blocks of code that are only compiled under certain circumstances. For example, we could have a block that is only used for debugging:

[C]

Listing 2345: main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int func(int x)
5  {
6      return x % 4;
7  }
8
9  int main()
10 {
11     int a, b;
12
13     a = 10;
14     b = func(a);
15
16     #ifdef DEBUG
17         printf("func(%d) => %d\n", a, b);
18     #endif
19
20     return 0;
21 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Debug_Code_C
MD5: 4daa8123f7112e7487ab54f16f80d34b

Here, the block indicated by the `DEBUG` flag is only included in the build if we define this preprocessing flag, which is what we expect for a debug version of the build. In the release version, however, we want to keep debug information out of the build, so we don't use this flag during the build process.

Ada doesn't define a preprocessor as part of the language. Some Ada toolchains — like the GNAT toolchain — do have a preprocessor that could create code similar to the one we've just seen. When programming in Ada, however, the recommendation is to use configuration packages to select code blocks that are meant to be included in the application.

When using a configuration package, the example above can be written as:

[Ada]

Listing 2346: config.ads

```
1  package Config is
2
3      Debug : constant Boolean := False;
4
5  end Config;
```

Listing 2347: func.ads

```
1  function Func (X : Integer) return Integer;
```

Listing 2348: func.adb

```

1 function Func (X : Integer) return Integer is
2 begin
3     return X mod 4;
4 end Func;

```

Listing 2349: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Config;
3 with Func;
4
5 procedure Main is
6     A, B : Integer;
7 begin
8     A := 10;
9     B := Func (A);
10
11     if Config.Debug then
12         Put_Line ("Func(" & Integer'Image (A) & ") => "
13                 & Integer'Image (B));
14     end if;
15 end Main;

```

Code block metadata

```

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Debug_Code_Ada
MD5: b643b683098fa7ad5568a69c9f2c000f

```

In this example, Config is a configuration package. The version of Config we're seeing here is the release version. The debug version of the Config package looks like this:

```

package Config is
    Debug : constant Boolean := True;
end Config;

```

The compiler makes sure to remove dead code. In the case of the release version, since Config.Debug is constant and set to **False**, the compiler is smart enough to remove the call to Put_Line from the build.

As you can see, both versions of Config are very similar to each other. The general idea is to create packages that declare the same constants, but using different values.

In C, we differentiate between the debug and release versions by selecting the appropriate preprocessing flags, but in Ada, we select the appropriate configuration package during the build process. Since the file name is usually the same (config.ads for the example above), we may want to store them in distinct directories. For the example above, we could have:

- src/debug/config.ads for the debug version, and
- src/release/config.ads for the release version.

Then, we simply select the appropriate configuration package for each version of the build by indicating the correct path to it. When using **GPRbuild**, we can select the appropriate directory where the config.ads file is located. We can use scenario variables in our project, which allow for creating different versions of a build. For example:

```
project Default is
```

(continues on next page)

(continued from previous page)

```

type Mode_Type is ("debug", "release");

Mode : Mode_Type := external ("mode", "debug");

for Source_Dirs use ("src", "src/" & Mode);
for Object_Dir use "obj";
for Main use ("main.adb");

end Default;
```

In this example, we're defining a scenario type called `Mode_Type`. Then, we're declaring the scenario variable `Mode` and using it in the `Source_Dirs` declaration to complete the path to the subdirectory containing the `config.ads` file. The expression `"src/" & Mode` concatenates the user-specified mode to select the appropriate subdirectory.

We can then set the mode on the command-line. For example:

```
gprbuild -P default.gpr -Xmode=release
```

In addition to selecting code blocks for the build, we could also specify values that depend on the target build. For our example above, we may want to create two versions of the application, each one having a different version of a `MOD_VALUE` that is used in the implementation of `func()`. In C, we can achieve this by using preprocessing flags and defining the corresponding version in `APP_VERSION`. Then, depending on the value of `APP_VERSION`, we define the corresponding value of `MOD_VALUE`.

[C]

Listing 2350: defs.h

```

1  #ifndef APP_VERSION
2  #define APP_VERSION    1
3  #endif
4
5  #if APP_VERSION == 1
6  #define MOD_VALUE      4
7  #endif
8
9  #if APP_VERSION == 2
10 #define MOD_VALUE      5
11 #endif
```

Listing 2351: main.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "defs.h"
5
6  int func(int x)
7  {
8      return x % MOD_VALUE;
9  }
10
11 int main()
12 {
13     int a, b;
14
15     a = 10;
16     b = func(a);
17
```

(continues on next page)

(continued from previous page)

```

18   return 0;
19 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.App_Version_C
MD5: 9f204dcc65b70618324c48be0dbdffbe

If not defined outside, the code above will compile version #1 of the application. We can change this by specifying a value for APP_VERSION during the build (e.g. as a Makefile switch).

For the Ada version of this code, we can create two configuration packages for each version of the application. For example:

[Ada]

Listing 2352: app_defs.ads

```

1  -- ./src/app_1/app_defs.ads
2
3  package App_Defs is
4
5      Mod_Value : constant Integer := 4;
6
7  end App_Defs;

```

Listing 2353: func.ads

```

1  function Func (X : Integer) return Integer;

```

Listing 2354: func.adb

```

1  with App_Defs;
2
3  function Func (X : Integer) return Integer is
4  begin
5      return X mod App_Defs.Mod_Value;
6  end Func;

```

Listing 2355: main.adb

```

1  with Func;
2
3  procedure Main is
4      A, B : Integer;
5  begin
6      A := 10;
7      B := Func (A);
8  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.App_Version_Ada
MD5: 7c8e4280e74c04ab51073b25e8f53995

The code above shows the version #1 of the configuration package. The corresponding implementation for version #2 looks like this:


```
-- ./src/app_2/app_defs.ads

package App_Defs is

    Mod_Value : constant Integer := 5;

end App_Defs;
```

Again, we just need to select the appropriate configuration package for each version of the build, which we can easily do when using **GPRbuild**.

87.3 Handling variability & reusability dynamically

87.3.1 Records with discriminants

In basic terms, records with discriminants are records that include "parameters" in their type definitions. This allows for adding more flexibility to the type definition. In the section about *pointers* (page 1872), we've seen this example:

[Ada]

Listing 2356: main.adb

```
1 procedure Main is
2     type Arr is array (Integer range <>) of Integer;
3
4     type S (Last : Positive) is record
5         A : Arr (0 .. Last);
6     end record;
7
8     V : S (9);
9 begin
10     null;
11 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Rec_Disc_Ada
MD5: 02fa8fa7832a262b99aee139a1b5b7a6

Here, Last is the discriminant for type S. When declaring the variable V as S (9), we specify the actual index of the last position of the array component A by setting the Last discriminant to 9.

We can create an equivalent implementation in C by declaring a **struct** with a pointer to an array:

[C]

Listing 2357: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct {
5     int * a;
6     const int last;
7 } S;
8
9 S init_s (int last)
10 {
```

(continues on next page)

(continued from previous page)

```

11     S v = { malloc (sizeof(int) * last + 1), last };
12     return v;
13 }
14
15 int main(int argc, const char * argv[])
16 {
17     S v = init_s (9);
18
19     return 0;
20 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Rec_Disc_C
MD5: 8f8b53c38c2ef8c1624208a2d8fd13ef

Here, we need to explicitly allocate the a array of the S struct via a call to `malloc()`, which allocates memory space on the heap. In the Ada version, in contrast, the array (V.A) is allocated on the stack and we don't need to explicitly allocate it.

Note that the information that we provide as the discriminant to the record type (in the Ada code) is constant, so we cannot assign a value to it. For example, we cannot write:

[Ada]

```
V.Last := 10;      -- COMPILATION ERROR!
```

In the C version, we declare the `last` field constant to get the same behavior.

[C]

```
v.last = 10;      // COMPILATION ERROR!
```

Note that the information provided as discriminants is visible. In the example above, we could display `Last` by writing:

[Ada]

```
Put_Line ("Last : " & Integer'Image (V.Last));
```

Also note that, even if a type is private, we can still access the information of the discriminants if they are visible in the *public* part of the type declaration. Let's rewrite the example above:

[Ada]

Listing 2358: array_definition.ads

```

1  package Array_Definition is
2      type Arr is array (Integer range <>) of Integer;
3
4      type S (Last : Integer) is private;
5
6  private
7      type S (Last : Integer) is record
8          A : Arr (0 .. Last);
9      end record;
10
11 end Array_Definition;

```

Listing 2359: main.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Array_Definition; use Array_Definition;
3
4 procedure Main is
5   V : S (9);
6 begin
7   Put_Line ("Last : " & Integer'Image (V.Last));
8 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Rec_Disc_Ada_Private
MD5: fa0158c3c61dd9ec7e4000416672f9e9

Build output

main.adb:5:04: warning: variable "V" is read but never assigned [-gnatwv]

Runtime output

Last : 9

Even though the S type is now private, we can still display Last because this discriminant is visible in the *non-private* part of package Array_Definition.

87.3.2 Variant records

In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a **case**. This is the general structure:

[Ada]

```
type Var_Rec (V : F) is record
    case V is
        when Opt_1 => F1 : Type_1;
        when Opt_2 => F2 : Type_2;
    end case;
end record;
```

Let's look at this example:

[Ada]

Listing 2360: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5   type Float_Int (Use_Float : Boolean) is record
6     case Use_Float is
7       when True  => F : Float;
8       when False => I : Integer;
9     end case;
10  end record;
11
12  procedure Display (V : Float_Int) is
```

(continues on next page)

(continued from previous page)

```

13   begin
14       if V.Use_Float then
15           Put_Line ("Float value: " & Float'Image (V.F));
16       else
17           Put_Line ("Integer value: " & Integer'Image (V.I));
18       end if;
19   end Display;
20
21   F : constant Float_Int := (Use_Float => True,  F => 10.0);
22   I : constant Float_Int := (Use_Float => False, I => 9);
23
24   begin
25       Display (F);
26       Display (I);
27   end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Var_Rec_Ada
MD5: 72dd64c22d65fc527af0c3de73ff7966

Runtime output

```

Float value: 1.00000E+01
Integer value: 9

```

Here, we declare F containing a floating-point value, and I containing an integer value. In the Display procedure, we present the correct information to the user according to the Use_Float discriminant of the Float_Int type.

We can implement this example in C by using unions:

[C]

Listing 2361: main.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5      int use_float;
6      union {
7          float f;
8          int i;
9      };
10 } float_int;
11
12 float_int init_float (float f)
13 {
14     float_int v;
15
16     v.use_float = 1;
17     v.f         = f;
18     return v;
19 }
20
21 float_int init_int (int i)
22 {
23     float_int v;
24
25     v.use_float = 0;
26     v.i         = i;

```

(continues on next page)

(continued from previous page)

```

27     return v;
28 }
29
30 void display (float_int v)
31 {
32     if (v.use_float) {
33         printf("Float value   : %f\n", v.f);
34     }
35     else {
36         printf("Integer value : %d\n", v.i);
37     }
38 }
39
40 int main(int argc, const char * argv[])
41 {
42     float_int f = init_float (10.0);
43     float_int i = init_int (9);
44
45     display (f);
46     display (i);
47
48     return 0;
49 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Var_Rec_C
MD5: ac0ad1e6ff7f2154e9dbb6838999a62e

Runtime output

```

Float value   : 10.000000
Integer value : 9

```

Similar to the Ada code, we declare `f` containing a floating-point value, and `i` containing an integer value. One difference is that we use the `init_float()` and `init_int()` functions to initialize the `float_int` struct. These functions initialize the correct field of the union and set the `use_float` field accordingly.

Variant records and unions

There is, however, a difference in accessibility between variant records in Ada and unions in C. In C, we're allowed to access any field of the union regardless of the initialization:

[C]

```

float_int v = init_float (10.0);

printf("Integer value : %d\n", v.i);

```

This feature is useful to create overlays. In this specific example, however, the information displayed to the user doesn't make sense, since the union was initialized with a floating-point value (`v.f`) and, by accessing the integer field (`v.i`), we're displaying it as if it was an integer value.

In Ada, accessing the wrong component would raise an exception at run-time ("discriminant check failed"), since the component is checked before being accessed:

[Ada]

```

V : constant Float_Int := (Use_Float => True, F => 10.0);
begin
  Put_Line ("Integer value: " & Integer'Image (V.I));
  -- ^ Constraint_Error is raised!

```

Using this method prevents wrong information being used in other parts of the program.

To get the same behavior in Ada as we do in C, we need to explicitly use the `Unchecked_Union` aspect in the type declaration. This is the modified example:

[Ada]

Listing 2362: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4
5   type Float_Int_Union (Use_Float : Boolean) is record
6     case Use_Float is
7       when True  => F : Float;
8       when False => I : Integer;
9     end case;
10  end record
11  with Unchecked_Union;
12
13  V : constant Float_Int_Union := (Use_Float => True, F => 10.0);
14
15 begin
16   Put_Line ("Integer value: " & Integer'Image (V.I));
17 end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Unchecked_Union_Ada
MD5: f6c5eacbd96c23531d02bb47a9668ac5

Runtime output

Integer value: 1092616192

Now, we can display the integer component (`V.I`) even though we initialized the floating-point component (`V.F`). As expected, the information displayed by the test application in this case doesn't make sense.

Note that, when using the `Unchecked_Union` aspect in the declaration of a variant record, the reference discriminant is not available anymore, since it isn't stored as part of the record. Therefore, we cannot access the `Use_Float` discriminant as in the following code:

[Ada]

```

V : constant Float_Int_Union := (Use_Float => True, F => 10.0);
begin
  if V.Use_Float then      -- COMPILATION ERROR!
    -- Do something...
  end if;

```

Unchecked unions are particularly useful in Ada when creating bindings for C code.

Optional components

We can also use variant records to specify optional components of a record. For example:
[Ada]

Listing 2363: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr is array (Integer range <>) of Integer;
5
6   type Extra_Info is (No, Yes);
7
8   type S_Var (Last : Integer; Has_Extra_Info : Extra_Info) is record
9     A : Arr (0 .. Last);
10
11     case Has_Extra_Info is
12       when No    => null;
13       when Yes   => B : Arr (0 .. Last);
14     end case;
15   end record;
16
17   V1 : S_Var (Last => 9, Has_Extra_Info => Yes);
18   V2 : S_Var (Last => 9, Has_Extra_Info => No);
19 begin
20   Put_Line ("Size of V1 is: " & Integer'Image (V1'Size));
21   Put_Line ("Size of V2 is: " & Integer'Image (V2'Size));
22 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Var_Rec_Null_Ada
MD5: 548235fa8458302ba025c8fa49e61777

Build output

```
main.adb:17:04: warning: variable "V1" is read but never assigned [-gnatwv]
main.adb:18:04: warning: variable "V2" is read but never assigned [-gnatwv]
```

Runtime output

```
Size of V1 is: 704
Size of V2 is: 384
```

Here, in the declaration of `S_Var`, we don't have any component in case `Has_Extra_Info` is false. The component is simply set to `null` in this case.

When running the example above, we see that the size of `V1` is greater than the size of `V2` due to the extra `B` component — which is only included when `Has_Extra_Info` is true.

Optional output information

We can use optional components to prevent subprograms from generating invalid information that could be misused by the caller. Consider the following example:

[C]

Listing 2364: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

(continues on next page)

(continued from previous page)

```

3
4 float calculate (float f1,
5                 float f2,
6                 int *success)
7 {
8     if (f1 < f2) {
9         *success = 1;
10        return f2 - f1;
11    }
12    else {
13        *success = 0;
14        return 0.0;
15    }
16 }
17
18 void display (float v,
19             int success)
20 {
21     if (success) {
22         printf("Value = %f\n", v);
23     }
24     else {
25         printf("Calculation error!\n");
26     }
27 }
28
29 int main(int argc, const char * argv[])
30 {
31     float f;
32     int success;
33
34     f = calculate (1.0, 0.5, &success);
35     display (f, success);
36
37     f = calculate (0.5, 1.0, &success);
38     display (f, success);
39
40     return 0;
41 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Non_Opt_C
MD5: 56f8a72782c4a54d8a6026aa39ce421a

Runtime output

```

Calculation error!
Value = 0.500000

```

In this code, we're using the output parameter `success` of the `calculate()` function to indicate whether the calculation was successful or not. This approach has a major problem: there's no way to prevent that the invalid value returned by `calculate()` in case of an error is misused in another computation. For example:

[C]

```

int main(int argc, const char * argv[])
{
    float f;
    int success;

```

(continues on next page)

(continued from previous page)

```
f = calculate (1.0, 0.5, &success);

f = f * 0.25;    // Using f in another computation even though
                 // calculate() returned a dummy value due to error!
                 // We should have evaluated "success", but we didn't.

return 0;
}
```

We cannot prevent access to the returned value or, at least, force the caller to evaluate success before using the returned value.

This is the corresponding code in Ada:

[Ada]

Listing 2365: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      function Calculate (F1, F2 : Float;
6                          Success : out Boolean) return Float is
7      begin
8          if F1 < F2 then
9              Success := True;
10             return F2 - F1;
11         else
12             Success := False;
13             return 0.0;
14         end if;
15     end Calculate;
16
17     procedure Display (V : Float; Success : Boolean) is
18     begin
19         if Success then
20             Put_Line ("Value = " & Float'Image (V));
21         else
22             Put_Line ("Calculation error!");
23         end if;
24     end Display;
25
26     F      : Float;
27     Success : Boolean;
28 begin
29     F := Calculate (1.0, 0.5, Success);
30     Display (F, Success);
31
32     F := Calculate (0.5, 1.0, Success);
33     Display (F, Success);
34 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Non_Opt_Ada
MD5: bb27fd31660ad604487f908934a3d3cb

Runtime output

```
Calculation error!
Value = 5.00000E-01
```

The Ada code above suffers from the same drawbacks as the C code. Again, there's no way to prevent misuse of the invalid value returned by Calculate in case of errors.

However, in Ada, we can use variant records to make the component unavailable and therefore prevent misuse of this information. Let's rewrite the original example and *wrap* the returned value in a variant record:

[Ada]

Listing 2366: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      type Opt_Float (Success : Boolean) is record
6          case Success is
7              when False => null;
8              when True  => F : Float;
9          end case;
10     end record;
11
12     function Calculate (F1, F2 : Float) return Opt_Float is
13     begin
14         if F1 < F2 then
15             return (Success => True, F => F2 - F1);
16         else
17             return (Success => False);
18         end if;
19     end Calculate;
20
21     procedure Display (V : Opt_Float) is
22     begin
23         if V.Success then
24             Put_Line ("Value = " & Float'Image (V.F));
25         else
26             Put_Line ("Calculation error!");
27         end if;
28     end Display;
29
30     begin
31         Display (Calculate (1.0, 0.5));
32         Display (Calculate (0.5, 1.0));
33     end Main;

```

Code block metadata

```
Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Opt_Ada
MD5: 8b70cd16d5ff13611567fa71059d6891
```

Runtime output

```
Calculation error!
Value = 5.00000E-01
```

In this example, we can determine whether the calculation was successful or not by evaluating the Success component of the Opt_Float. If the calculation wasn't successful, we won't be able to access the F component of the Opt_Float. As mentioned before, trying to access the component in this case would raise an exception. Therefore, in case of errors, we can ensure that no information is misused after the call to Calculate.

87.3.3 Object orientation

In the *previous section* (page 1910), we've seen that we can add variability to records by using discriminants. Another approach is to use *tagged* records, which are the base for object-oriented programming in Ada.

Type extension

A tagged record type is declared by adding the **tagged** keyword. For example:

[Ada]

Listing 2367: main.adb

```

1  procedure Main is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      type Tagged_Rec is tagged record
8          V : Integer;
9      end record;
10
11     R1 : Rec;
12     R2 : Tagged_Rec;
13
14     pragma Unreferenced (R1, R2);
15 begin
16     R1 := (V => 0);
17     R2 := (V => 0);
18 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Tagged_Type_Decl
MD5: 53810d3bb5aa7e7b1483270d974eb025

In this simple example, there isn't much difference between the Rec and Tagged_Rec type. However, tagged types can be derived and extended. For example:

[Ada]

Listing 2368: main.adb

```

1  procedure Main is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      -- We cannot declare this:
8      --
9      -- type Ext_Rec is new Rec with record
10     --     V : Integer;
11     -- end record;
12
13     type Tagged_Rec is tagged record
14         V : Integer;
15     end record;
16
17     -- But we can declare this:
18     --
```

(continues on next page)

(continued from previous page)

```

19  type Ext_Tagged_Rec is new Tagged_Rec with record
20      V2 : Integer;
21  end record;
22
23      R1 : Rec;
24      R2 : Tagged_Rec;
25      R3 : Ext_Tagged_Rec;
26
27  pragma Unreferenced (R1, R2, R3);
28  begin
29      R1 := (V => 0);
30      R2 := (V => 0);
31      R3 := (V => 0, V2 => 0);
32  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Tagged_Type_Extension_Decl
MD5: 707a3e6b220357f50f6792190b000c91

As indicated in the example, a type derived from an untagged type cannot have an extension. The compiler indicates this error if you uncomment the declaration of the Ext_Rec type above. In contrast, we can extend a tagged type, as we did in the declaration of Ext_Tagged_Rec. In this case, Ext_Tagged_Rec has all the components of the Tagged_Rec type (V, in this case) plus the additional components from its own type declaration (V2, in this case).

Overriding subprograms

Previously, we've seen that subprograms can be overridden. For example, if we had implemented a Reset and a Display procedure for the Rec type that we declared above, these procedures would be available for an Ext_Rec type derived from Rec. Also, we could override these procedures for the Ext_Rec type. In Ada, we don't need object-oriented programming features to do that: simple (untagged) records can be used to derive types, inherit operations and override them. However, in applications where the actual subprogram to be called is determined dynamically at run-time, we need dispatching calls. In this case, we must use tagged types to implement this.

Comparing untagged and tagged types

Let's discuss the similarities and differences between untagged and tagged types based on this example:

[Ada]

Listing 2369: p.ads

```

1  package P is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      procedure Display (R : Rec);
8      procedure Reset (R : out Rec);
9
10     type New_Rec is new Rec;
11
12     overriding procedure Display (R : New_Rec);

```

(continues on next page)

(continued from previous page)

```

13  not overriding procedure New_Op (R : in out New_Rec);
14
15  type Tagged_Rec is tagged record
16      V : Integer;
17  end record;
18
19  procedure Display (R : Tagged_Rec);
20  procedure Reset (R : out Tagged_Rec);
21
22  type Ext_Tagged_Rec is new Tagged_Rec with record
23      V2 : Integer;
24  end record;
25
26  overriding procedure Display (R : Ext_Tagged_Rec);
27  overriding procedure Reset (R : out Ext_Tagged_Rec);
28  not overriding procedure New_Op (R : in out Ext_Tagged_Rec);
29
30  end P;
```

Listing 2370: p.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
4
5      procedure Display (R : Rec) is
6      begin
7          Put_Line ("TYPE: REC");
8          Put_Line ("Rec.V = " & Integer'Image (R.V));
9          New_Line;
10     end Display;
11
12     procedure Reset (R : out Rec) is
13     begin
14         R.V := 0;
15     end Reset;
16
17     procedure Display (R : New_Rec) is
18     begin
19         Put_Line ("TYPE: NEW_REC");
20         Put_Line ("New_Rec.V = " & Integer'Image (R.V));
21         New_Line;
22     end Display;
23
24     procedure New_Op (R : in out New_Rec) is
25     begin
26         R.V := R.V + 1;
27     end New_Op;
28
29     procedure Display (R : Tagged_Rec) is
30     begin
31         -- Using External_Tag attribute to retrieve the tag as a string
32         Put_Line ("TYPE: " & Tagged_Rec'External_Tag);
33         Put_Line ("Tagged_Rec.V = " & Integer'Image (R.V));
34         New_Line;
35     end Display;
36
37     procedure Reset (R : out Tagged_Rec) is
38     begin
39         R.V := 0;
40     end Reset;
```

(continues on next page)

(continued from previous page)

```

41
42 procedure Display (R : Ext_Tagged_Rec) is
43 begin
44   -- Using External_Tag attribute to retrieve the tag as a string
45   Put_Line ("TYPE: " & Ext_Tagged_Rec'External_Tag);
46   Put_Line ("Ext_Tagged_Rec.V = " & Integer'Image (R.V));
47   Put_Line ("Ext_Tagged_Rec.V2 = " & Integer'Image (R.V2));
48   New_Line;
49 end Display;
50
51 procedure Reset (R : out Ext_Tagged_Rec) is
52 begin
53   Tagged_Rec (R).Reset;
54   R.V2 := 0;
55 end Reset;
56
57 procedure New_Op (R : in out Ext_Tagged_Rec) is
58 begin
59   R.V := R.V + 1;
60 end New_Op;
61
62 end P;
```

Listing 2371: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with P;           use P;
3
4 procedure Main is
5   X_Rec          : Rec;
6   X_New_Rec      : New_Rec;
7
8   X_Tagged_Rec   : aliased Tagged_Rec;
9   X_Ext_Tagged_Rec : aliased Ext_Tagged_Rec;
10
11   X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
12                       := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
13 begin
14   --
15   -- Reset all objects
16   --
17   Reset (X_Rec);
18   Reset (X_New_Rec);
19   X_Tagged_Rec.Reset; -- we could write "Reset (X_Tagged_Rec)" as well
20   X_Ext_Tagged_Rec.Reset;
21
22   --
23   -- Use new operations when available
24   --
25   New_Op (X_New_Rec);
26   X_Ext_Tagged_Rec.New_Op;
27
28   --
29   -- Display all objects
30   --
31   Display (X_Rec);
32   Display (X_New_Rec);
33   X_Tagged_Rec.Display; -- we could write "Display (X_Tagged_Rec)" as well
34   X_Ext_Tagged_Rec.Display;
35
36   --
```

(continues on next page)

(continued from previous page)

```
37  -- Resetting and display objects of Tagged_Rec'Class
38  --
39  Put_Line ("Operations on Tagged_Rec'Class");
40  Put_Line ("-----");
41  for E of X_Tagged_Rec_Array loop
42      E.Reset;
43      E.Display;
44  end loop;
45  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Tagged_Type_Extension_Decl
MD5: 29412b74db6680f0a0986b62e5284cf7

Runtime output

```
TYPE: REC
Rec.V = 0

TYPE: NEW_REC
New_Rec.V = 1

TYPE: P.TAGGED_REC
Tagged_Rec.V = 0

TYPE: P.EXT_TAGGED_REC
Ext_Tagged_Rec.V = 1
Ext_Tagged_Rec.V2 = 0

Operations on Tagged_Rec'Class
-----
TYPE: P.TAGGED_REC
Tagged_Rec.V = 0

TYPE: P.EXT_TAGGED_REC
Ext_Tagged_Rec.V = 0
Ext_Tagged_Rec.V2 = 0
```

These are the similarities between untagged and tagged types:

- We can derive types and inherit operations in both cases.
 - Both `X_New_Rec` and `X_Ext_Tagged_Rec` inherit the `Display` and `Reset` procedures from their respective ancestors.
- We can override operations in both cases.
- We can implement new operations in both cases.
 - Both `X_New_Rec` and `X_Ext_Tagged_Rec` implement a procedure called `New_Op`, which is not available for their respective ancestors.

Now, let's look at the differences between untagged and tagged types:

- We can dispatch calls for a given type class.
 - This is what we do when we iterate over objects of the `Tagged_Rec` class — in the loop over `X_Tagged_Rec_Array` at the last part of the `Main` procedure.
- We can use the dot notation.
 - We can write both `E.Reset` or `Reset (E)` forms: they're equivalent.

Dispatching calls

Let's look more closely at the dispatching calls implemented above. First, we declare the `X_Tagged_Rec_Array` array and initialize it with the access to objects of both parent and derived tagged types:

[Ada]

```
X_Tagged_Rec      : aliased Tagged_Rec;
X_Ext_Tagged_Rec  : aliased Ext_Tagged_Rec;

X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
                    := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
```

Here, we use the **aliased** keyword to be able to get access to the objects (via the `'Access` attribute).

Then, we loop over this array and call the `Reset` and `Display` procedures:

[Ada]

```
for E of X_Tagged_Rec_Array loop
    E.Reset;
    E.Display;
end loop;
```

Since we're using dispatching calls, the actual procedure that is selected depends on the type of the object. For the first element (`X_Tagged_Rec_Array (1)`), this is `Tagged_Rec`, while for the second element (`X_Tagged_Rec_Array (2)`), this is `Ext_Tagged_Rec`.

Dispatching calls are only possible for a type class — for example, the `Tagged_Rec'Class`. When the type of an object is known at compile time, the calls won't dispatch at runtime. For example, the call to the `Reset` procedure of the `X_Ext_Tagged_Rec` object (`X_Ext_Tagged_Rec.Reset`) will always take the overridden `Reset` procedure of the `Ext_Tagged_Rec` type. Similarly, if we perform a view conversion by writing `Tagged_Rec (A_Ext_Tagged_Rec).Display`, we're instructing the compiler to interpret `A_Ext_Tagged_Rec` as an object of type `Tagged_Rec`, so that the compiler selects the `Display` procedure of the `Tagged_Rec` type.

Interfaces

Another useful feature of object-oriented programming is the use of interfaces. In this case, we can define abstract operations, and implement them in the derived tagged types. We declare an interface by simply writing **type T is interface**. For example:

[Ada]

```
type My_Interface is interface;

procedure Op (Obj : My_Interface) is abstract;

-- We cannot declare actual objects of an interface:
--
-- Obj : My_Interface; -- ERROR!
```

All operations on an interface type are abstract, so we need to write **is abstract** in the signature — as we did in the declaration of `Op` above. Also, since interfaces are abstract types and don't have an actual implementation, we cannot declare objects for it.

We can derive tagged types from an interface and implement the actual operations of that interface:

[Ada]


```
type My_Derived is new My_Interface with null record;

procedure Op (Obj : My_Derived);
```

Note that we're not using the **tagged** keyword in the declaration because any type derived from an interface is automatically tagged.

Let's look at an example with an interface and two derived tagged types:

[Ada]

Listing 2372: p.ads

```
1 package P is
2
3     type Display_Interface is interface;
4     procedure Display (D : Display_Interface) is abstract;
5
6     type Small_Display_Type is new Display_Interface with null record;
7     procedure Display (D : Small_Display_Type);
8
9     type Big_Display_Type is new Display_Interface with null record;
10    procedure Display (D : Big_Display_Type);
11
12 end P;
```

Listing 2373: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5     procedure Display (D : Small_Display_Type) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Using Small_Display_Type");
9     end Display;
10
11    procedure Display (D : Big_Display_Type) is
12        pragma Unreferenced (D);
13    begin
14        Put_Line ("Using Big_Display_Type");
15    end Display;
16
17 end P;
```

Listing 2374: main.adb

```
1 with P; use P;
2
3 procedure Main is
4     D_Small : Small_Display_Type;
5     D_Big   : Big_Display_Type;
6
7     procedure Dispatching_Display (D : Display_Interface'Class) is
8     begin
9         D.Display;
10    end Dispatching_Display;
11
12 begin
13     Dispatching_Display (D_Small);
14     Dispatching_Display (D_Big);
```

(continues on next page)

(continued from previous page)

```
15 end Main;
```

Code block metadata

```
Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Interfaces_1
MD5: 564eba158b2f8fc3efea9e892a21caa9
```

Runtime output

```
Using Small_Display_Type
Using Big_Display_Type
```

In this example, we have an interface type `Display_Interface` and two tagged types that are derived from `Display_Interface`: `Small_Display_Type` and `Big_Display_Type`.

Both types (`Small_Display_Type` and `Big_Display_Type`) implement the interface by overriding the `Display` procedure. Then, in the inner procedure `Dispatching_Display` of the `Main` procedure, we perform a dispatching call depending on the actual type of `D`.

Deriving from multiple interfaces

We may derive a type from multiple interfaces by simply writing `type Derived_T is new T1 and T2 with null record`. For example:

[Ada]

Listing 2375: transceivers.ads

```
1 package Transceivers is
2
3     type Send_Interface is interface;
4
5     procedure Send (Obj : in out Send_Interface) is abstract;
6
7     type Receive_Interface is interface;
8
9     procedure Receive (Obj : in out Receive_Interface) is abstract;
10
11     type Transceiver is new Send_Interface and Receive_Interface
12         with null record;
13
14     procedure Send (D : in out Transceiver);
15     procedure Receive (D : in out Transceiver);
16
17 end Transceivers;
```

Listing 2376: transceivers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Transceivers is
4
5     procedure Send (D : in out Transceiver) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Sending data...");
9     end Send;
10
11     procedure Receive (D : in out Transceiver) is
12         pragma Unreferenced (D);
```

(continues on next page)

(continued from previous page)

```
13   begin
14       Put_Line ("Receiving data...");
15   end Receive;
16
17 end Transceivers;
```

Listing 2377: main.adb

```
1 with Transceivers; use Transceivers;
2
3 procedure Main is
4     D : Transceiver;
5     begin
6         D.Send;
7         D.Receive;
8     end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Multiple_Interfaces
MD5: c81813941bd3458eaf7b1fd39b010a03

Runtime output

Sending data...
Receiving data...

In this example, we're declaring two interfaces (Send_Interface and Receive_Interface) and the tagged type Transceiver that derives from both interfaces. Since we need to implement the interfaces, we implement both Send and Receive for Transceiver.

Abstract tagged types

We may also declare abstract tagged types. Note that, because the type is abstract, we cannot use it to declare objects for it — this is the same as for interfaces. We can only use it to derive other types. Let's look at the abstract tagged type declared in the Abstract_Transceivers package:

[Ada]

Listing 2378: abstract_transceivers.ads

```
1 with Transceivers; use Transceivers;
2
3 package Abstract_Transceivers is
4
5     type Abstract_Transceiver is abstract new Send_Interface and
6         Receive_Interface with null record;
7
8     procedure Send (D : in out Abstract_Transceiver);
9     -- We don't implement Receive for Abstract_Transceiver!
10
11 end Abstract_Transceivers;
```

Listing 2379: abstract_transceivers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Abstract_Transceivers is
4
```

(continues on next page)

(continued from previous page)

```

5  procedure Send (D : in out Abstract_Transceiver) is
6      pragma Unreferenced (D);
7  begin
8      Put_Line ("Sending data...");
9  end Send;
10
11 end Abstract_Transceivers;

```

Listing 2380: main.adb

```

1  with Abstract_Transceivers; use Abstract_Transceivers;
2
3  procedure Main is
4      D : Abstract_Transceiver;
5  begin
6      D.Send;
7      D.Receive;
8  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Multiple_Interfaces
MD5: c2b0b3aab1ffc9c3b9a0749bf6721088

Build output

```

main.adb:4:09: error: type of object cannot be abstract
main.adb:7:06: error: call to abstract procedure must be dispatching
gprbuild: *** compilation phase failed

```

In this example, we declare the abstract tagged type `Abstract_Transceiver`. Here, we're only partially implementing the interfaces from which this type is derived: we're implementing `Send`, but we're skipping the implementation of `Receive`. Therefore, `Receive` is an abstract operation of `Abstract_Transceiver`. Since any tagged type that has abstract operations is abstract, we must indicate this by adding the **abstract** keyword in type declaration.

Also, when compiling this example, we get an error because we're trying to declare an object of `Abstract_Transceiver` (in the `Main` procedure), which is not possible. Naturally, if we derive another type from `Abstract_Transceiver` and implement `Receive` as well, then we can declare objects of this derived type. This is what we do in the `Full_Transceivers` below:

[Ada]

Listing 2381: full_transceivers.ads

```

1  with Abstract_Transceivers; use Abstract_Transceivers;
2
3  package Full_Transceivers is
4
5      type Full_Transceiver is new Abstract_Transceiver with null record;
6      procedure Receive (D : in out Full_Transceiver);
7
8  end Full_Transceivers;

```

Listing 2382: full_transceivers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```
3 package body Full_Transceivers is
4
5     procedure Receive (D : in out Full_Transceiver) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Receiving data...");
9     end Receive;
10
11 end Full_Transceivers;
```

Listing 2383: main.adb

```
1 with Full_Transceivers; use Full_Transceivers;
2
3 procedure Main is
4     D : Full_Transceiver;
5 begin
6     D.Send;
7     D.Receive;
8 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Multiple_Interfaces
MD5: 77a86a6d917547d306a89422e7522111

Runtime output

Sending data...
Receiving data...

Here, we implement the Receive procedure for the Full_Transceiver. Therefore, the type doesn't have any abstract operation, so we can use it to declare objects.

From simple derivation to OOP

In the [section about simple derivation](#) (page 1898), we've seen an example where the actual selection was done at *implementation* time by renaming one of the packages:

[Ada]

```
with Drivers_1;

package Drivers renames Drivers_1;
```

Although this approach is useful in many cases, there might be situations where we need to select the actual driver dynamically at runtime. Let's look at how we could rewrite that example using interfaces, tagged types and dispatching calls:

[Ada]

Listing 2384: drivers_base.ads

```
1 package Drivers_Base is
2
3     type Transceiver is interface;
4
5     procedure Send (Device : Transceiver; Data : Integer) is abstract;
6     procedure Receive (Device : Transceiver; Data : out Integer) is abstract;
7     procedure Display (Device : Transceiver) is abstract;
```

(continues on next page)

(continued from previous page)

```

8
9 end Drivers_Base;

```

Listing 2385: drivers_1.ads

```

1 with Drivers_Base;
2
3 package Drivers_1 is
4
5     type Transceiver is new Drivers_Base.Transceiver with null record;
6
7     procedure Send (Device : Transceiver; Data : Integer);
8     procedure Receive (Device : Transceiver; Data : out Integer);
9     procedure Display (Device : Transceiver);
10
11 end Drivers_1;

```

Listing 2386: drivers_1.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Drivers_1 is
4
5     procedure Send (Device : Transceiver; Data : Integer) is null;
6
7     procedure Receive (Device : Transceiver; Data : out Integer) is
8         pragma Unreferenced (Device);
9     begin
10         Data := 42;
11     end Receive;
12
13     procedure Display (Device : Transceiver) is
14         pragma Unreferenced (Device);
15     begin
16         Put_Line ("Using Drivers_1");
17     end Display;
18
19 end Drivers_1;

```

Listing 2387: drivers_2.ads

```

1 with Drivers_Base;
2
3 package Drivers_2 is
4
5     type Transceiver is new Drivers_Base.Transceiver with null record;
6
7     procedure Send (Device : Transceiver; Data : Integer);
8     procedure Receive (Device : Transceiver; Data : out Integer);
9     procedure Display (Device : Transceiver);
10
11 end Drivers_2;

```

Listing 2388: drivers_2.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Drivers_2 is
4
5     procedure Send (Device : Transceiver; Data : Integer) is null;

```

(continues on next page)

(continued from previous page)

```
6
7  procedure Receive (Device : Transceiver; Data : out Integer) is
8      pragma Unreferenced (Device);
9  begin
10     Data := 7;
11 end Receive;
12
13 procedure Display (Device : Transceiver) is
14     pragma Unreferenced (Device);
15 begin
16     Put_Line ("Using Drivers_2");
17 end Display;
18
19 end Drivers_2;
```

Listing 2389: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Drivers_Base;
4  with Drivers_1;
5  with Drivers_2;
6
7  procedure Main is
8      D1 : aliased Drivers_1.Transceiver;
9      D2 : aliased Drivers_2.Transceiver;
10     D   : access Drivers_Base.Transceiver'Class;
11
12     I   : Integer;
13
14     type Driver_Number is range 1 .. 2;
15
16     procedure Select_Driver (N : Driver_Number) is
17     begin
18         if N = 1 then
19             D := D1'Access;
20         else
21             D := D2'Access;
22         end if;
23         D.Display;
24     end Select_Driver;
25
26 begin
27     Select_Driver (1);
28     D.Send (999);
29     D.Receive (I);
30     Put_Line (Integer'Image (I));
31
32     Select_Driver (2);
33     D.Send (999);
34     D.Receive (I);
35     Put_Line (Integer'Image (I));
36 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Tagged_Drivers
MD5: d823b7231f1adf003fb6f545cb482308

Runtime output

```

Using Drivers_1
42
Using Drivers_2
7

```

In this example, we declare the Transceiver interface in the Drivers_Base package. This interface is then used to derive the tagged types Transceiver from both Drivers_1 and Drivers_2 packages.

In the Main procedure, we use the access to Transceiver'Class — from the interface declared in the Drivers_Base package — to declare D. This object D contains the access to the actual driver loaded at any specific time. We select the driver at runtime in the inner Select_Driver procedure, which initializes D (with the access to the selected driver). Then, any operation on D triggers a dispatching call to the selected driver.

Further resources

In the appendices, we have a step-by-step [hands-on overview of object-oriented programming](#) (page 1965) that discusses how to translate a simple system written in C to an equivalent system in Ada using object-oriented programming.

87.3.4 Pointer to subprograms

Pointers to subprograms allow us to dynamically select an appropriate subprogram at runtime. This selection might be triggered by an external event, or simply by the user. This can be useful when multiple versions of a routine exist, and the decision about which one to use cannot be made at compilation time.

This is an example on how to declare and use pointers to functions in C:

[C]

Listing 2390: main.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void show_msg_v1 (char *msg)
5  {
6      printf("Using version #1: %s\n", msg);
7  }
8
9  void show_msg_v2 (char *msg)
10 {
11     printf("Using version #2:\n %s\n", msg);
12 }
13
14 int main()
15 {
16     int selection = 1;
17     void (*current_show_msg) (char *);
18
19     switch (selection)
20     {
21         case 1: current_show_msg = &show_msg_v1;    break;
22         case 2: current_show_msg = &show_msg_v2;    break;
23         default: current_show_msg = NULL;           break;
24     }
25
26     if (current_show_msg != NULL)
27     {

```

(continues on next page)

(continued from previous page)

```

28     current_show_msg ("Hello there!");
29 }
30 else
31 {
32     printf("ERROR: no version of show_msg() selected!\n");
33 }
34
35 return 0;
36 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Selecting_Subprogram_C
MD5: 414c99fca2490611d20d031f8549ff59

Runtime output

Using version #1: Hello there!

The example above contains two versions of the `show_msg()` function: `show_msg_v1()` and `show_msg_v2()`. The function is selected depending on the value of `selection`, which initializes the function pointer `current_show_msg`. If there's no corresponding value, `current_show_msg` is set to `null` — alternatively, we could have selected a default version of `show_msg()` function. By calling `current_show_msg ("Hello there!")`, we're calling the function that `current_show_msg` is pointing to.

This is the corresponding implementation in Ada:

[Ada]

Listing 2391: `show_subprogram_selection.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Subprogram_Selection is
4
5      procedure Show_Msg_V1 (Msg : String) is
6      begin
7          Put_Line ("Using version #1: " & Msg);
8      end Show_Msg_V1;
9
10     procedure Show_Msg_V2 (Msg : String) is
11     begin
12         Put_Line ("Using version #2: ");
13         Put_Line (Msg);
14     end Show_Msg_V2;
15
16     type Show_Msg_Proc is access procedure (Msg : String);
17
18     Current_Show_Msg : Show_Msg_Proc;
19     Selection        : Natural;
20
21 begin
22     Selection := 1;
23
24     case Selection is
25         when 1 => Current_Show_Msg := Show_Msg_V1'Access;
26         when 2 => Current_Show_Msg := Show_Msg_V2'Access;
27         when others => Current_Show_Msg := null;
28     end case;
29

```

(continues on next page)

(continued from previous page)

```

30  if Current_Show_Msg /= null then
31      Current_Show_Msg ("Hello there!");
32  else
33      Put_Line ("ERROR: no version of Show_Msg selected!");
34  end if;
35
36  end Show_Subprogram_Selection;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Selecting_Subprogram_Ada
 MD5: ee41e042e3b879b4a2671bfe6d8072aa

Runtime output

Using version #1: Hello there!

The structure of the code above is very similar to the one used in the C code. Again, we have two version of Show_Msg: Show_Msg_V1 and Show_Msg_V2. We set Current_Show_Msg according to the value of Selection. Here, we use 'Access to get access to the corresponding procedure. If no version of Show_Msg is available, we set Current_Show_Msg to null.

Pointers to subprograms are also typically used as callback functions. This approach is extensively used in systems that process events, for example. Here, we could have a two-layered system:

- A layer of the system (an event manager) triggers events depending on information from sensors.
 - For each event, callback functions can be registered.
 - The event manager calls registered callback functions when an event is triggered.
- Another layer of the system registers callback functions for specific events and decides what to do when those events are triggered.

This approach promotes information hiding and component decoupling because:

- the layer of the system responsible for managing events doesn't need to know what the callback function actually does, while
- the layer of the system that implements callback functions remains agnostic to implementation details of the event manager — for example, how events are implemented in the event manager.

Let's see an example in C where we have a process_values() function that calls a callback function (process_one) to process a list of values:

[C]

Listing 2392: process_values.h

```

1  typedef int (*process_one_callback) (int);
2
3  void process_values (int          *values,
4                      int          len,
5                      process_one_callback process_one);

```

Listing 2393: process_values.c

```

1  #include "process_values.h"
2

```

(continues on next page)

(continued from previous page)

```
3  #include <assert.h>
4  #include <stdio.h>
5
6  void process_values (int          *values,
7                      int          len,
8                      process_one_callback process_one)
9  {
10     int i;
11
12     assert (process_one != NULL);
13
14     for (i = 0; i < len; i++)
15     {
16         values[i] = process_one (values[i]);
17     }
18 }
```

Listing 2394: main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "process_values.h"
5
6  int proc_10 (int val)
7  {
8     return val + 10;
9  }
10
11 # define LEN_VALUES      5
12
13 int main()
14 {
15
16     int values[LEN_VALUES] = { 1, 2, 3, 4, 5 };
17     int i;
18
19     process_values (values, LEN_VALUES, &proc_10);
20
21     for (i = 0; i < LEN_VALUES; i++)
22     {
23         printf("Value [%d] = %d\n", i, values[i]);
24     }
25
26     return 0;
27 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Callback_C
MD5: ff5c8611d0901f40b6c4a9effeb0a323

Runtime output

```
Value [0] = 11
Value [1] = 12
Value [2] = 13
Value [3] = 14
Value [4] = 15
```

As mentioned previously, `process_values()` doesn't have any knowledge about what pro-

cess_one() does with the integer value it receives as a parameter. Also, we could replace proc_10() by another function without having to change the implementation of process_values().

Note that process_values() calls an assert() for the function pointer to compare it against null. Here, instead of checking the validity of the function pointer, we're expecting the caller of process_values() to provide a valid pointer.

This is the corresponding implementation in Ada:

[Ada]

Listing 2395: values_processing.ads

```

1 package Values_Processing is
2
3   type Integer_Array is array (Positive range <>) of Integer;
4
5   type Process_One_Callback is not null access
6     function (Value : Integer) return Integer;
7
8   procedure Process_Values (Values      : in out Integer_Array;
9                             Process_One :      Process_One_Callback);
10
11 end Values_Processing;
```

Listing 2396: values_processing.adb

```

1 package body Values_Processing is
2
3   procedure Process_Values (Values      : in out Integer_Array;
4                             Process_One :      Process_One_Callback) is
5   begin
6     for I in Values'Range loop
7       Values (I) := Process_One (Values (I));
8     end loop;
9   end Process_Values;
10
11 end Values_Processing;
```

Listing 2397: proc_10.ads

```

1 function Proc_10 (Value : Integer) return Integer;
```

Listing 2398: proc_10.adb

```

1 function Proc_10 (Value : Integer) return Integer is
2 begin
3   return Value + 10;
4 end Proc_10;
```

Listing 2399: show_callback.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Values_Processing; use Values_Processing;
4 with Proc_10;
5
6 procedure Show_Callback is
7   Values : Integer_Array := (1, 2, 3, 4, 5);
8 begin
9   Process_Values (Values, Proc_10'Access);
```

(continues on next page)

(continued from previous page)

```

10
11   for I in Values'Range loop
12       Put_Line ("Value ["
13               & Positive'Image (I)
14               & "] = "
15               & Integer'Image (Values (I)));
16   end loop;
17 end Show_Callback;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.Callback_Ada
MD5: f49c54f0d14193d305c0e962a392ab67

Runtime output

```

Value [ 1] = 11
Value [ 2] = 12
Value [ 3] = 13
Value [ 4] = 14
Value [ 5] = 15
```

Similar to the implementation in C, the `Process_Values` procedure receives the access to a callback routine, which is then called for each value of the `Values` array.

Note that the declaration of `Process_One_Callback` makes use of the **not null access** declaration. By using this approach, we ensure that any parameter of this type has a valid value, so we can always call the callback routine.

87.4 Design by components using dynamic libraries

In the previous sections, we have shown how to use packages to create separate components of a system. As we know, when designing a complex system, it is advisable to separate concerns into distinct units, so we can use Ada packages to represent each unit of a system. In this section, we go one step further and create separate dynamic libraries for each component, which we'll then link to the main application.

Let's suppose we have a main system (`Main_System`) and a component A (`Component_A`) that we want to use in the main system. For example:

[Ada]

Listing 2400: component_a.ads

```

1  --
2  --   File: component_a.ads
3  --
4  package Component_A is
5
6      type Float_Array is array (Positive range <>) of Float;
7
8      function Average (Data : Float_Array) return Float;
9
10 end Component_A;
```

Listing 2401: component_a.adb

```

1  --
2  --   File: component_a.adb
```

(continues on next page)

(continued from previous page)

```

3  --
4  package body Component_A is
5
6      function Average (Data : Float_Array) return Float is
7          Total : Float := 0.0;
8      begin
9          for Value of Data loop
10             Total := Total + Value;
11          end loop;
12          return Total / Float (Data'Length);
13      end Average;
14
15 end Component_A;

```

Listing 2402: main_system.adb

```

1  --
2  -- File: main_system.adb
3  --
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  with Component_A; use Component_A;
7
8  procedure Main_System is
9      Values : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
10     Average_Value : Float;
11 begin
12     Average_Value := Average (Values);
13     Put_Line ("Average = " & Float'Image (Average_Value));
14 end Main_System;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Reusability.System_For_Dyn_Lib
MD5: d759132b787e636d4bcd5f8cd6393f2a

Runtime output

Average = 1.15000E+01

Note that, in the source-code example above, we're indicating the name of each file. We'll now see how to organize those files in a structure that is suitable for the GNAT build system (**GPRbuild**).

In order to discuss how to create dynamic libraries, we need to dig into some details about the build system. With GNAT, we can use project files for **GPRbuild** to easily design dynamic libraries. Let's say we use the following directory structure for the code above:

```

|- component_a
|   | component_a.gpr
|   |   |- src
|   |       | component_a.adb
|   |       | component_a.ads
|- main_system
|   | main_system.gpr
|   |   |- src
|   |       | main_system.adb

```

Here, we have two directories: *component_a* and *main_system*. Each directory contains a project file (with the *.gpr* file extension) and a source-code directory (*src*).

In the source-code example above, we've seen the content of files *component_a.ads*,

component_a.adb and main_system.adb. Now, let's discuss how to write the project file for Component_A (component_a.gpr), which will build the dynamic library for this component:

```
library project Component_A is

  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Create_Missing_Dirs use "True";
  for Library_Name use "component_a";
  for Library_Kind use "dynamic";
  for Library_Dir use "lib";

end Component_A;
```

The project is defined as a *library project* instead of *project*. This tells **GPRbuild** to build a library instead of an executable binary. We then specify the library name using the *Library_Name* attribute, which is required, so it must appear in a library project. The next two library-related attributes are optional, but important for our use-case. We use:

- *Library_Kind* to specify that we want to create a dynamic library — by default, this attribute is set to *static*;
- *Library_Dir* to specify the directory where the library is stored.

In the project file of our main system (main_system.gpr), we just need to reference the project of Component_A using a *with* clause and indicating the correct path to that project file:

```
with "../component_a/component_a.gpr";

project Main_System is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Create_Missing_Dirs use "True";
  for Main use ("main_system.adb");
end Main_System;
```

GPRbuild takes care of selecting the correct settings to link the dynamic library created for Component_A with the main application (Main_System) and build an executable.

We can use the same strategy to create a Component_B and dynamically link to it in the Main_System. We just need to create the separate structure for this component — with the appropriate Ada packages and project file — and include it in the project file of the main system using a *with* clause:

```
with "../component_a/component_a.gpr";
with "../component_b/component_b.gpr";

...
```

Again, **GPRbuild** takes care of selecting the correct settings to link both dynamic libraries together with the main application.

You can find more details and special setting for library projects in the [GPRbuild documentation](https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects)⁴⁴⁸.

In the GNAT toolchain

The GNAT toolchain includes a more advanced example focusing on how to load dynamic libraries at runtime. You can find it in the share/examples/gnat/plugins directory of the GNAT toolchain installation. As described in the README file from that directory, this

⁴⁴⁸ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects

example "comprises a main program which probes regularly for the existence of shared libraries in a known location. If such libraries are present, it uses them to implement features initially not present in the main program."

PERFORMANCE CONSIDERATIONS

88.1 Overall expectations

All in all, there should not be significant performance differences between code written in Ada and code written in C, provided that they are semantically equivalent. Taking the current GNAT implementation and its GCC C counterpart for example, most of the code generation and optimization phases are shared between C and Ada — so there's not one compiler more efficient than the other. Furthermore, the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. They should be equivalent on average.

When comparing the performance of C and Ada code, differences might be observed. This usually comes from the fact that, while the two piece *appear* semantically equivalent, they happen to be actually quite different; C code semantics do not implicitly apply the same run-time checks that Ada does. This section will present common ways for improving Ada code performance.

88.2 Switches and optimizations

Clever use of compilation switches might optimize the performance of an application significantly. In this section, we'll briefly look into some of the switches available in the GNAT toolchain.

88.2.1 Optimizations levels

Optimization levels can be found in many compilers for multiple languages. On the lowest level, the GNAT compiler doesn't optimize the code at all, while at the higher levels, the compiler analyses the code and optimizes it by removing unnecessary operations and making the most use of the target processor's capabilities.

By being part of GCC, GNAT offers the same `-O_` switches as GCC:

Switc	Description
-O0	No optimization: the generated code is completely unoptimized. This is the default optimization level.
-O1	Moderate optimization.
-O2	Full optimization.
-O3	Same optimization level as for -O2. In addition, further optimization strategies, such as aggressive automatic inlining and vectorization.

Note that the higher the level, the longer the compilation time. For fast compilation during development phase, unless you're working on benchmarking algorithms, using `-O0` is probably a good idea.

In addition to the levels presented above, GNAT also has the `-Os` switch, which allows for optimizing code and data usage.

88.2.2 Inlining

As we've seen in the previous section, automatic inlining depends on the optimization level. The highest optimization level (`-O3`), for example, performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (`-O2`). Therefore, the general recommendation is to not *just* select `-O3` for the optimized version of an application, but instead compare it the optimized version built with `-O2`.

In some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the `Inline` aspect. Let's reuse an example from a previous chapter and inline the `Average` function:

[Ada]

Listing 2403: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is array (Positive range <>) of Float;
4
5     function Average (Data : Float_Array) return Float
6         with Inline;
7
8 end Float_Arrays;
```

Listing 2404: float_arrays.adb

```
1 package body Float_Arrays is
2
3     function Average (Data : Float_Array) return Float is
4         Total : Float := 0.0;
5     begin
6         for Value of Data loop
7             Total := Total + Value;
8         end loop;
9         return Total / Float (Data'Length);
10    end Average;
11
12 end Float_Arrays;
```

Listing 2405: compute_average.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Arrays; use Float_Arrays;
4
5 procedure Compute_Average is
6     Values      : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
7     Average_Value : Float;
8 begin
9     Average_Value := Average (Values);
10    Put_Line ("Average = " & Float'Image (Average_Value));
11 end Compute_Average;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Inlining
 MD5: faf9d0d8cd5aefd7a48bcd950b1256fa

Runtime output

Average = 1.15000E+01

When compiling this example, GNAT will inline Average in the Compute_Average procedure.

In order to effectively use this aspect, however, we need to set the optimization level to at least -O1 and use the -gnatn switch, which instructs the compiler to take the Inline aspect into account.

Note, however, that the Inline aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram. In this case, we get a compilation warning from GNAT.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

In addition to the Inline aspect, we also have the Inline_Always aspect. In contrast to the former aspect, however, the Inline_Always aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the Inline aspect, there might be situations where a subprogram has the Inline_Always aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

88.3 Checks and assertions

88.3.1 Checks

Ada provides many runtime checks to ensure that the implementation is working as expected. For example, when accessing an array, we would like to make sure that we're not accessing a memory position that is not allocated for that array. This is achieved by an index check.

Another example of runtime check is the verification of valid ranges. For example, when adding two integer numbers, we would like to ensure that the result is still in the valid range — that the value is neither too large nor too small. This is achieved by a range check. Likewise, arithmetic operations shouldn't overflow or underflow. This is achieved by an overflow check.

Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots. For example, checking the index of an array in a sorting algorithm may significantly decrease its performance. In those cases, suppressing the check may be an option. We can achieve this suppression by using `pragma Suppress` (Index_Check). For example:

[Ada]

```
procedure Sort (A : in out Integer_Array) is
  pragma Suppress (Index_Check);
begin
```

(continues on next page)

(continued from previous page)

```
-- (implementation removed...)
null;
end Sort;
```

In case of overflow checks, we can use `pragma Suppress` (`Overflow_Check`) to suppress them:

```
function Some_Computation (A, B : Int32) return Int32 is
  pragma Suppress (Overflow_Check);
begin
  -- (implementation removed...)
  null;
end Sort;
```

We can also deactivate overflow checks for integer types using the `-gnato` switch when compiling a source-code file with GNAT. In this case, overflow checks in the whole file are deactivated.

It is also possible to suppress all checks at once using `pragma Suppress` (`All_Checks`). In addition, GNAT offers a compilation switch called `-gnatp`, which has the same effect on the whole file.

Note, however, that this kind of suppression is just a recommendation to the compiler. There's no guarantee that the compiler will actually suppress any of the checks because the compiler may not be able to do so — typically because the hardware happens to do it. For example, if the machine traps on any access via address zero, requesting the removal of null access value checks in the generated code won't prevent the checks from happening.

It is important to differentiate between required and redundant checks. Let's consider the following example in C:

[C]

Listing 2406: main.c

```
1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int a = 8, b = 0, res;
6
7      res = a / b;
8
9      // printing the result
10     printf("res = %d\n", res);
11
12     return 0;
13 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C.Dev.Performance.Division_By_Zero_C
MD5: c8d95cbdd76618108119886c27ce7eb6

Because C doesn't have language-defined checks, as soon as the application tries to divide a value by zero in `res = a / b`, it'll break — on Linux, for example, you may get the following error message by the operating system: Floating point exception (core dumped). Therefore, we need to manually introduce a check for zero before this operation. For example:

[C]

Listing 2407: main.c

```

1  #include <stdio.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int a = 8, b = 0, res;
6
7      if (b != 0) {
8          res = a / b;
9
10         // printing the result
11         printf("res = %d\n", res);
12     }
13     else
14     {
15         // printing error message
16         printf("Error: cannot calculate value (division by zero)\n");
17     }
18
19     return 0;
20 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Division_By_Zero_Check_C
MD5: 67ea0140d8248674b4aac06825c7cdbe

Runtime output

Error: cannot calculate value (division by zero)

This is the corresponding code in Ada:

[Ada]

Listing 2408: show_division_by_zero.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Division_By_Zero is
4      A   : Integer := 8;
5      B   : Integer := 0;
6      Res : Integer;
7  begin
8      Res := A / B;
9
10     Put_Line ("Res = " & Integer'Image (Res));
11 end Show_Division_By_Zero;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Division_By_Zero_Ada
MD5: 2af6690eb977203ef7ce2178d15255af

Build output

```

show_division_by_zero.adb:8:15: warning: division by zero [enabled by default]
show_division_by_zero.adb:8:15: warning: Constraint_Error will be raised at run_
↳time [enabled by default]

```

Runtime output

```
raised CONSTRAINT_ERROR : show_division_by_zero.adb:8 divide by zero
```

Similar to the first version of the C code, we're not explicitly checking for a potential division by zero here. In Ada, however, this check is *automatically inserted* by the language itself. When running the application above, an exception is raised when the application tries to divide the value in A by zero. We could introduce exception handling in our example, so that we get the same message as we did in the second version of the C code:

[Ada]

Listing 2409: show_division_by_zero.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Division_By_Zero is
4   A   : Integer := 8;
5   B   : Integer := 0;
6   Res : Integer;
7 begin
8   Res := A / B;
9
10  Put_Line ("Res = " & Integer'Image (Res));
11 exception
12   when Constraint_Error =>
13     Put_Line ("Error: cannot calculate value (division by zero)");
14   when others =>
15     null;
16 end Show_Division_By_Zero;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Division_By_Zero_Check_Ada
MD5: a96a94c15fda5f6c5feb232d615blea3

Build output

```
show_division_by_zero.adb:8:15: warning: division by zero [enabled by default]
show_division_by_zero.adb:8:15: warning: Constraint_Error will be raised at run_
↳time [enabled by default]
```

Runtime output

```
Error: cannot calculate value (division by zero)
```

This example demonstrates that the division check for `Res := A / B` is required and shouldn't be suppressed. In contrast, a check is redundant — and therefore not required — when we know that the condition that leads to a failure can never happen. In many cases, the compiler itself detects redundant checks and eliminates them (for higher optimization levels). Therefore, when improving the performance of your application, you should:

1. keep all checks active for most parts of the application;
2. identify the hot-spots of your application;
3. identify which checks haven't been eliminated by the optimizer on these hot-spots;
4. identify which of those checks are redundant;
5. only suppress those checks that are redundant, and keep the required ones.

88.3.2 Assertions

We've already discussed assertions in *this section of the SPARK chapter* (page 1852). Assertions are user-defined checks that you can add to your code using the `pragma Assert`. For example:

[Ada]

```
function Some_Computation (A, B : Int32) return Int32 is
  Res : Int32;
begin
  -- (implementation removed...)

  pragma Assert (Res >= 0);

  return Res;
end Sort;
```

Assertions that are specified with `pragma Assert` are not enabled by default. You can enable them by setting the assertion policy to *check* — using `pragma Assertion_Policy (Check)` — or by using the `-gnata` switch when compiling with GNAT.

Similar to the checks discussed previously, assertions can generate significant overhead when used at hot-spots. Restricting those assertions to development (e.g. debug version) and turning them off on the release version may be an option. In this case, formal proof — as discussed in the *SPARK chapter* (page 1845) — can help you. By formally proving that assertions will never fail at run-time, you can safely deactivate them.

88.4 Dynamic vs. static structures

Ada generally speaking provides more ways than C or C++ to write simple dynamic structures, that is to say structures that have constraints computed after variables. For example, it's quite typical to have initial values in record types:

[Ada]

```
type R is record
  F : Some_Field := Call_To_Some_Function;
end record;
```

However, the consequences of the above is that any declaration of a instance of this type without an explicit value for `F` will issue a call to `Call_To_Some_Function`. More subtle issue may arise with elaboration. For example, it's possible to write:

Listing 2410: some_functions.ads

```
1 package Some_Functions is
2
3   function Some_Function_Call return Integer is (2);
4
5   function Some_Other_Function_Call return Integer is (10);
6
7 end Some_Functions;
```

Listing 2411: values.ads

```
1 with Some_Functions; use Some_Functions;
2
3 package Values is
4   A_Start : Integer := Some_Function_Call;
```

(continues on next page)

(continued from previous page)

```
5   A_End    : Integer := Some_Other_Function_Call;
6 end Values;
```

Listing 2412: arr_def.ads

```
1 with Values; use Values;
2
3 package Arr_Def is
4   type Arr is array (Integer range A_Start .. A_End) of Integer;
5 end Arr_Def;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Dynamic_Array
MD5: 0c97cecb64d27e935724c8b5f941fb4f

It may indeed be appealing to be able to change the values of A_Start and A_End at startup so as to align a series of arrays dynamically. The consequence, however, is that these values will not be known statically, so any code that needs to access to boundaries of the array will need to read data from memory. While it's perfectly fine most of the time, there may be situations where performances are so critical that static values for array boundaries must be enforced.

Here's a last case which may also be surprising:

[Ada]

Listing 2413: arr_def.ads

```
1 package Arr_Def is
2   type Arr is array (Integer range <>) of Integer;
3
4   type R (D1, D2 : Integer) is record
5     F1 : Arr (1 .. D1);
6     F2 : Arr (1 .. D2);
7   end record;
8 end Arr_Def;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Record_With_Arrays
MD5: e7b2656433279d36db87506276b68398

In the code above, R contains two arrays, F1 and F2, respectively constrained by the discriminant D1 and D2. The consequence is, however, that to access F2, the run-time needs to know how large F1 is, which is dynamically constrained when creating an instance. Therefore, accessing to F2 requires a computation involving D1 which is slower than, let's say, two pointers in an C array that would point to two different arrays.

Generally speaking, when values are used in data structures, it's useful to always consider where they're coming from, and if their value is static (computed by the compiler) or dynamic (only known at run-time). There's nothing fundamentally wrong with dynamically constrained types, unless they appear in performance-critical pieces of the application.

88.5 Pointers vs. data copies

In the section about *pointers* (page 1872), we mentioned that the Ada compiler will automatically pass parameters by reference when needed. Let's look into what "when needed" means. The fundamental point to understand is that the parameter types determine how

the parameters are passed in and/or out. The parameter modes do not control how parameters are passed.

Specifically, the language standards specifies that scalar types are always passed by value, and that some other types are always passed by reference. It would not make sense to make a copy of a task when passing it as a parameter, for example. So parameters that can be passed reasonably by value will be, and those that must be passed by reference will be. That's the safest approach.

But the language also specifies that when the parameter is an array type or a record type, and the record/array components are all by-value types, then the compiler decides: it can pass the parameter using either mechanism. The critical case is when such a parameter is large, e.g., a large matrix. We don't want the compiler to pass it by value because that would entail a large copy, and indeed the compiler will not do so. But if the array or record parameter is small, say the same size as an address, then it doesn't matter how it is passed and by copy is just as fast as by reference. That's why the language gives the choice to the compiler. Although the language does not mandate that large parameters be passed by reference, any reasonable compiler will do the right thing.

The modes do have an effect, but not in determining how the parameters are passed. Their effect, for parameters passed by value, is to determine how many times the value is copied. For mode **in** and mode **out** there is just one copy. For mode **in out** there will be two copies, one in each direction.

Therefore, unlike C, you don't have to use access types in Ada to get better performance when passing arrays or records to subprograms. The compiler will almost certainly do the right thing for you.

Let's look at this example:

[C]

Listing 2414: main.c

```

1  #include <stdio.h>
2
3  struct Data {
4      int prev, curr;
5  };
6
7  void update(struct Data *d,
8             int v)
9  {
10     d->prev = d->curr;
11     d->curr = v;
12 }
13
14 void display(const struct Data *d)
15 {
16     printf("Prev : %d\n", d->prev);
17     printf("Curr : %d\n", d->curr);
18 }
19
20 int main(int argc, const char * argv[])
21 {
22     struct Data D1 = { 0, 1 };
23
24     update (&D1, 3);
25     display (&D1);
26
27     return 0;
28 }
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Passing_Rec_By_Reference_C
MD5: 9087e26168e49d095b5e0776d6330d69

Runtime output

```
Prev : 1  
Curr : 3
```

In this C code example, we're using pointers to pass D1 as a reference to update and display. In contrast, the equivalent code in Ada simply uses the parameter modes to specify the data flow directions. The mechanisms used to pass the values do not appear in the source code.

[Ada]

Listing 2415: update_record.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 procedure Update_Record is  
4  
5     type Data is record  
6         Prev : Integer;  
7         Curr : Integer;  
8     end record;  
9  
10    procedure Update (D : in out Data;  
11                     V : Integer) is  
12    begin  
13        D.Prev := D.Curr;  
14        D.Curr := V;  
15    end Update;  
16  
17    procedure Display (D : Data) is  
18    begin  
19        Put_Line ("Prev: " & Integer'Image (D.Prev));  
20        Put_Line ("Curr: " & Integer'Image (D.Curr));  
21    end Display;  
22  
23    D1 : Data := (0, 1);  
24  
25    begin  
26        Update (D1, 3);  
27        Display (D1);  
28    end Update_Record;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Passing_Rec_By_Reference_Ada
MD5: 6c64fb73e2cf490c0a129f0cd73c190b

Runtime output

```
Prev: 1  
Curr: 3
```

In the calls to Update and Display, D1 is always be passed by reference. Because no extra copy takes place, we get a performance that is equivalent to the C version. If we had used arrays in the example above, D1 would have been passed by reference as well:

[Ada]

Listing 2416: update_array.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Update_Array is
4
5      type Data_State is (Prev, Curr);
6      type Data is array (Data_State) of Integer;
7
8      procedure Update (D : in out Data;
9                       V : Integer) is
10     begin
11         D (Prev) := D (Curr);
12         D (Curr) := V;
13     end Update;
14
15     procedure Display (D : Data) is
16     begin
17         Put_Line ("Prev: " & Integer'Image (D (Prev)));
18         Put_Line ("Curr: " & Integer'Image (D (Curr)));
19     end Display;
20
21     D1 : Data := (0, 1);
22
23 begin
24     Update (D1, 3);
25     Display (D1);
26 end Update_Array;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Passing_Array_By_Reference_Ada
MD5: 5fb27811f34543fc4150eb4fddbe7034

Runtime output

```

Prev: 1
Curr: 3

```

Again, no extra copy is performed in the calls to `Update` and `Display`, which gives us optimal performance when dealing with arrays and avoids the need to use access types to optimize the code.

88.5.1 Function returns

Previously, we've discussed the cost of passing complex records as arguments to subprograms. We've seen that we don't have to use explicit access type parameters to get better performance in Ada. In this section, we'll briefly discuss the cost of function returns.

In general, we can use either procedures or functions to initialize a data structure. Let's look at this example in C:

[C]

Listing 2417: main.c

```

1  #include <stdio.h>
2
3  struct Data {
4      int prev, curr;
5  };

```

(continues on next page)

(continued from previous page)

```

6
7 void init_data(struct Data *d)
8 {
9     d->prev = 0;
10    d->curr = 1;
11 }
12
13 struct Data get_init_data()
14 {
15     struct Data d = { 0, 1 };
16
17     return d;
18 }
19
20 int main(int argc, const char * argv[])
21 {
22     struct Data D1;
23
24     D1 = get_init_data();
25
26     init_data(&D1);
27
28     return 0;
29 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Init_Rec_Proc_And_Func_C
MD5: 0586636d5e25c0d6bec2257af75ae998

This code example contains two subprograms that initialize the Data structure:

- `init_data()`, which receives the data structure as a reference (using a pointer) and initializes it, and
- `get_init_data()`, which returns the initialized structure.

In C, we generally avoid implementing functions such as `get_init_data()` because of the extra copy that is needed for the function return.

This is the corresponding implementation in Ada:

[Ada]

Listing 2418: `init_record.adb`

```

1 procedure Init_Record is
2
3     type Data is record
4         Prev : Integer;
5         Curr : Integer;
6     end record;
7
8     procedure Init (D : out Data) is
9     begin
10        D := (Prev => 0, Curr => 1);
11    end Init;
12
13    function Init return Data is
14        D : constant Data := (Prev => 0, Curr => 1);
15    begin
16        return D;
17    end Init;

```

(continues on next page)

(continued from previous page)

```

18
19     D1 : Data;
20
21     pragma Unreferenced (D1);
22 begin
23     D1 := Init;
24
25     Init (D1);
26 end Init_Record;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Init_Rec_Proc_And_Func_Ada
MD5: 0f930eea432a82d78840b72c0714b283

Build output

```

init_record.adb:25:10: warning: aspect Unreferenced specified for "D1" [enabled by ↵
↵ default]

```

In this example, we have two versions of `Init`: one using a procedural form, and the other one using a functional form. Note that, because of Ada's support for subprogram overloading, we can use the same name for both subprograms.

The issue is that assignment of a function result entails a copy, just as if we assigned one variable to another. For example, when assigning a function result to a constant, the function result is copied into the memory for the constant. That's what is happening in the above examples for the initialized variables.

Therefore, in terms of performance, the same recommendations apply: for large types we should avoid writing functions like the `Init` function above. Instead, we should use the procedural form of `Init`. The reason is that the compiler necessarily generates a copy for the `Init` function, while the `Init` procedure uses a reference for the output parameter, so that the actual record initialization is performed in place in the caller's argument.

An exception to this is when we use functions returning values of limited types, which by definition do not allow assignment. Here, to avoid allowing something that would otherwise look suspiciously like an assignment, the compiler generates the function body so that it builds the result directly into the object being assigned. No copy takes place.

We could, for example, rewrite the example above using limited types:

[Ada]

Listing 2419: `init_limited_record.adb`

```

1  procedure Init_Limited_Record is
2
3      type Data is limited record
4          Prev : Integer;
5          Curr : Integer;
6      end record;
7
8      function Init return Data is
9      begin
10         return D : Data do
11             D.Prev := 0;
12             D.Curr := 1;
13         end return;
14     end Init;
15
16     D1 : Data := Init;

```

(continues on next page)

(continued from previous page)

```
17
18     pragma Unreferenced (D1);
19 begin
20     null;
21 end Init_Limited_Record;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.Performance.Init_Lim_Rec_Proc_And_Func_Ada
MD5: 57fc1b3f69b42dd4633b0c67e252c2d2

In this example, `D1 : Data := Init;` has the same cost as the call to the procedural form — `Init (D1);` — that we've seen in the previous example. This is because the assignment is done in place.

Note that limited types require the use of the extended return statements (`return ... do ... end return`) in function implementations. Also note that, because the `Data` type is limited, we can only use the `Init` function in the declaration of `D1`; a statement in the code such as `D1 := Init;` is therefore forbidden.

ARGUMENTATION AND BUSINESS PERSPECTIVES

The technical benefits of a migration from C to Ada are usually relatively straightforward to demonstrate. Hopefully, this course provides a good basis for it. However, when faced with an actual business decision to make, additional considerations need to be taken into account, such as return on investment, perennity of the solution, tool support, etc. This section will cover a number of usual questions and provide elements of answers.

89.1 What's the expected ROI of a C to Ada transition?

Switching from one technology to another is a cost, may that be in terms of training, transition of the existing environment or acquisition of new tools. This investment needs to be matched with an expected return on investment, or ROI, to be consistent. Of course, it's incredibly difficult to provide a firm answer to how much money can be saved by transitioning, as this is highly dependent on specific project objectives and constraints. We're going to provide qualitative and quantitative arguments here, from the perspective of a project that has to reach a relatively high level of integrity, that is to say a system where the occurrence of a software failure is a relatively costly event.

From a qualitative standpoint, there are various times in the software development life cycle where defects can be found:

1. on the developer's desk
2. during component testing
3. during integration testing
4. after deployment
5. during maintenance

Numbers from studies vary greatly on the relative costs of defects found at each of these phases, but there's a clear ordering between them. For example, a defect found while developing is orders of magnitude less expensive to fix than a defect found e.g. at integration time, which may involve costly debugging sessions and slow down the entire system acceptance. The whole purpose of Ada and SPARK is to push defect detection to the developer's desk as much as possible; at least for all of these defects that can be identified at that level. While the strict act of writing software may be taking more effort because of all of the additional safeguards, this should have a significant and positive impact down the line and help to control costs overall. The exact value this may translate into is highly business dependent.

From a quantitative standpoint, two studies have been done almost 25 years apart and provide similar insights:

- Rational Software in 1995 found that the cost of developing software in Ada was overall half as much as the cost of developing software in C.
- VDC ran a study in 2018, finding that the cost savings of developing with Ada over C ranged from 6% to 38% in savings.

From a qualitative standpoint, in particular with regards to Ada and C from a formal proof perspective, an interesting presentation was made in 2017 by two researchers. They tried to apply formal proof on the same piece of code, developed in Ada/SPARK on one end and C/Frama-C on the other. Their results indicate that the Ada/SPARK technology is indeed more conducive to formal proof methodologies.

Although all of these studies have their own biases, they provide a good idea of what to expect in terms of savings once the initial investment in switching to Ada is made. This is assuming everything else is equal, in particular that the level of integrity is the same. In many situations, the migration to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will rise (it's more expensive to develop better software) and Ada is viewed as a means to mitigate this rise in development costs.

That being said, the point of this argument is not to say that it's not possible to write very safe and secure software with languages different than Ada. With the right expertise, the right processes and the right tools, it's done every day. The point is that Ada overall reduces the level of processes, expertise and tools necessary and will allow to reach the same target at a lower cost.

89.2 Who is using Ada today?

Ada was initially born as a DoD project, and thus got its initial customer base in aerospace and defence (A&D). At the time these lines are written and from the perspective of AdaCore, A&D is still the largest consumer of Ada today and covers about 70% of the market. This creates a consistent and long lasting set of established users as these project last often for decades, using the same codebase migrating from platform to platform.

More recently however, there has been an emerging interest for Ada in new communities of users such as automotive, medical device, industrial automation and overall cyber-security. This can probably be explained by a rise of safety, reliability and cyber-security requirements. The market is moving relatively rapidly today and we're anticipating an increase of the Ada footprint in these domains, while still remaining a technology of choice for the development of mission critical software.

89.3 What is the future of the Ada technology?

The first piece of the answer lies in the user base of the Ada language, as seen in the previous question. Projects using Ada in the aerospace and defence domain maintain source code over decades, providing healthy funding foundation for Ada-based technologies.

AdaCore being the author of this course, it's difficult for us to be fair in our description of other Ada compilation technologies. We will leave to the readers the responsibility of forging their own opinion. If they present a credible alternative to the GNAT compiler, then this whole section can be considered as void.

Assuming GNAT is the only option available, and acknowledging that this is an argument that we're hearing from a number of Ada adopters, let's discuss the "sole source" issue.

First of all, it's worth noting that industries are using a lot of software that is provided by only one source, so while non-ideal, these situations are also quite common.

In the case of the GNAT compiler however, while AdaCore is the main maintainer, this maintenance is done as part of an open-source community. This means that nothing prevents a third party to start selling a competing set of products based on the same compiler, provided that it too adopts the open-source approach. Our job is to be more cost-effective than the alternative, and indeed for the vast part this has prevented a competing offering to emerge. However, should AdaCore disappear or switch focus, Ada users would not be prevented from carrying on using its software (there is no lock) and a third party could take

over maintenance. This is not a theoretical case, this has been done in the past either by companies looking at supporting their own version of GNAT, vendors occupying a specific niche that was left uncovered, or hobbyists developing their own builds.

With that in mind, it's clear that the "sole source" provider issue is a circumstantial — nothing is preventing other vendors from emerging if the conditions are met.

89.4 Is the Ada toolset complete?

A language by itself is of little use for the development of safety-critical software. Instead, a complete toolset is needed to accompany the development process, in particular tools for edition, testing, static analysis, etc.

AdaCore provides a number of these tools either in through its core or add-on package. These include (as of 2019):

- An IDE (GNAT Studio)
- An Eclipse plug-in (GNATbench)
- A debugger (GDB)
- A testing tool (GNATtest)
- A structural code coverage tool (GNATcoverage)
- A metric computation tool (GNATmetric)
- A coding standard checker (GNATcheck)
- Static analysis tools (CodePeer, SPARK Pro)
- A Simulink code generator (QGen)
- An Ada parser to develop custom tools (libadalang)

Ada is, however, an internationally standardized language, and many companies are providing third party solutions to complete the toolset. Overall, the language can be and is used with tools on par with their equivalent C counterparts.

89.5 Where can I find Ada or SPARK developers?

A common question from teams on the verge of selecting Ada and SPARK is how to manage the developer team growth and turnover. While Ada and SPARK are taught by a growing number of universities worldwide, it may still be challenging to hire new staff with prior Ada experience.

Fortunately, Ada's base semantics are very close to those of C/C++, so that a good embedded software developer should be able to learn it relatively easily. This course is definitely a resource available to get started. Online training material is also available, together with on-site in person training.

In general, getting an engineer operational in Ada and SPARK shouldn't take more than a few weeks worth of time.

89.6 How to introduce Ada and SPARK in an existing code base?

The most common scenario when introducing Ada and SPARK to a project or a team is to do it within a pre-existing C codebase, which can already spread over hundreds of thousands if

not millions lines of code. Re-writing this software to Ada or SPARK is of course not practical and counterproductive.

Most teams select either a small piece of existing code which deserves particular attention, or new modules to develop, and concentrate on this. Developing this module or part of the application will also help in developing the coding patterns to be used for the particular project and company. This typically concentrates an effort of a few people on a few thousands lines of code. The resulting code can be linked to the rest of the C application. From there, the newly established practices and their benefit can slowly spread through the rest of the environment.

Establishing this initial core in Ada and SPARK is critical, and while learning the language isn't a particularly difficult task, applying it to its full capacity may require some expertise. One possibility to accelerate this initial process is to use AdaCore mentorship services.

CONCLUSION

Although Ada's syntax might seem peculiar to C developers at first glance, it was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner — as it is often the case in C.

Especially in the embedded domain, C developers are used to working at a very low level, which includes mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations because it was designed to replace Assembly language for faster, more efficient programming.

Ada can be used to describe high level semantics and architectures. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded Assembly code or bit-level data management. However, although Ada supports bitwise operations such as masks and shifts, they should be relatively rarely needed. When translating C code to Ada, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other. For example, we can interpret an arbitrary data type as a bit-field and perform low-level operations on it.

Because Ada is a strongly typed language, it doesn't define any implicit type conversions like C. If we try to compile Ada code that contains type mismatches, we'll get a compilation error. Because the compiler prevents mixing variables of different types without explicit type conversion, we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect. In this sense, Ada's type system encourages programmers to think about data at a high level of abstraction. Ada supports overlays and unchecked conversions as a way of converting between unrelated data type, which are typically used for interfacing with low-level elements such as registers.

In Ada, arrays aren't interchangeable with operations on pointers like in C. Also, array types are considered first-class citizens and have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and we can specify both the starting and ending bounds. In addition, Ada offers high-level operations for copying, slicing, and assigning values to arrays.

Although Ada supports pointers, most situations that would require a pointer in C do not in Ada. In the vast majority of the cases, indirect memory management can be hidden from the developer and thus prevent many potential errors. In C, pointers are typically used to pass references to subprograms, for example. In contrast, Ada parameter modes indicate the flow of information to the reader, leaving the means of passing that information to the compiler.

When translating pointers from C code to Ada, we need to assess whether they are needed in the first place. Ada pointers (access types) should only be used with complex structures that cannot be allocated at run-time. There are many situations that would require a pointer in C, but do not in Ada. For example, arrays — even when dynamically allocated —, results of functions, passing of large structures as parameters, access to registers, etc.

Because of the absence of namespaces, global names in C tend to be very long. Also, because of the absence of overloading, they can even encode type names in their name. In Ada, a package is a namespace. Also, we can use the private part of a package to declare private types and private subprograms. In fact, private types are useful for preventing the users of those types from depending on the implementation details. Another use-case is the prevention of package users from accessing the package state/data arbitrarily.

Ada has a dedicated set of features for interfacing with other languages, so we can easily interface with our existing C code before translating it to Ada. Also, GNAT includes automatic binding generators. Therefore, instead of re-writing the entire C code upfront, which isn't practical or cost-effective, we can selectively translate modules from C to Ada.

When it comes to implementing concurrency and real time, Ada offers several options. Ada provides high level constructs such as tasks and protected objects to express concurrency and synchronization, which can be used when running on top of an operating system such as Linux. On more constrained systems, such as bare metal or some real-time operating systems, a subset of the Ada tasking capabilities — known as the Ravenscar and Jorvik profiles — is available. Though restricted, this subset also has nice properties, in particular the absence of deadlock, the absence of priority inversion, schedulability and very small footprint. On bare metal systems, this also essentially means that Ada comes with its own real-time kernel. The advantage of using the full Ada tasking model or the restricted profiles is to enhance portability.

Ada includes many features typically used for embedded programming:

- Built-in support for handling interrupts, so we can process interrupts by attaching a handler — as a protected procedure — to it.
- Built-in support for handling both volatile and atomic data.
- Support for register overlays, which we can use to create a structure that facilitates manipulating bits from registers.
- Support for creating data streams for serialization of arbitrary information and transmission over a communication channel, such as a serial port.
- Built-in support for fixed-point arithmetic, which is an option when our target device doesn't have a floating-point unit or the result of calculations needs to be bit-exact.

Also, Ada compilers such as GNAT have built-in support for directly mixing Ada and Assembly code.

Ada also supports contracts, which can be associated with types and variables to refine values and define valid and invalid values. The most common kind of contract is a *range constraint* — using the **range** reserved word. Ada also supports contract-based programming in the form of preconditions and postconditions. One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations.

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. In C, variability is usually achieved through macros and function pointers, the former being tied to static variability and the latter to dynamic variability. Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. Examples of static variability in Ada are: genericity, simple derivation, configuration pragma files, and configuration packages. Examples of dynamic variability in Ada are: records with discriminants, variant records — which may include the use of unions —, object orientation, pointers to subprograms, and design by components using dynamic libraries.

There shouldn't be significant performance differences between code written in Ada and code written in C — provided that they are semantically equivalent. One reason is that the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. Therefore, they should be equivalent on average. However, when a piece of code in Ada is significantly slower than its counterpart in C, this usually comes from the fact that, while the two pieces of code appear

to be semantically equivalent, they happen to be actually quite different. Fortunately, there are strategies that we can use to improve the performance and make it equivalent to the C version. These are some examples:

- Clever use of compilation switches, which might optimize the performance of an application significantly.
- Suppression of checks at specific parts of the implementation.
 - Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots.
- Restriction of assertions to development code.
 - For example, we may use assertions in the debug version of the code and turn them off in the release version.
 - Also, we may use formal proof to decide which assertions we turn off in the release version. By formally proving that assertions will never fail at run-time, we can safely deactivate them.

Formal proof — a form of static analysis — can give strong guarantees about checks, for all possible conditions and all possible inputs. It verifies conditions prior to execution, even prior to compilation, so we can remove bugs earlier in the development phase. This is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **gnatprove** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof.

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering *exceptions* that interrupt the normal flow of execution. However, we may be able to prove that the language-defined checks won't raise exceptions at run-time. This is known as proving *Absence of Run-Time Errors*. Successful proof of these checks is highly significant in itself. One of the major resulting benefits is that we can deploy the final executable with checks disabled.

In many situations, the migration of C code to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will raise. However, Ada is a more expressive, powerful language, designed to reduce errors earlier in the life-cycle, thus reducing costs. Therefore, Ada makes it possible to write very safe and secure software at a lower cost than languages such as C.

APPENDIX A: HANDS-ON OBJECT-ORIENTED PROGRAMMING

The goal of this appendix is to present a hands-on view on how to translate a system from C to Ada and improve it with object-oriented programming.

91.1 System Overview

Let's start with an overview of a simple system that we'll implement and use below. The main system is called AB and it combines two systems A and B. System AB is not supposed to do anything useful. However, it can serve as a good model for the hands-on we're about to start.

This is a list of requirements for the individual systems A and B, and the combined system AB:

- System A:
 - The system can be activated and deactivated.
 - * During activation, the system's values are reset.
 - Its current value (in floating-point) can be retrieved.
 - * This value is the average of the two internal floating-point values.
 - Its current state (activated or deactivated) can be retrieved.
- System B:
 - The system can be activated and deactivated.
 - * During activation, the system's value is reset.
 - Its current value (in floating-point) can be retrieved.
 - Its current state (activated or deactivated) can be retrieved.
- System AB
 - The system contains an instance of system A and an instance of system B.
 - The system can be activated and deactivated.
 - * System AB activates both systems A and B during its own activation.
 - * System AB deactivates both systems A and B during its own deactivation.
 - Its current value (in floating-point) can be retrieved.
 - * This value is the average of the current values of systems A and B.
 - Its current state (activated or deactivated) can be retrieved.
 - * AB is only considered activated when both systems A and B are activated.

- The system's health can be checked.
 - * This check consists in calculating the absolute difference D between the current values of systems A and B and checking whether D is below a threshold of 0.1.

The source-code in the following section contains an implementation of these requirements.

91.2 Non Object-Oriented Approach

In this section, we look into implementations (in both C and Ada) of system AB that don't make use of object-oriented programming.

91.2.1 Starting point in C

Let's start with an implementation in C for the system described above:

[C]

Listing 2420: system_a.h

```
1 typedef struct {
2     float val[2];
3     int active;
4 } A;
5
6 void A_activate (A *a);
7
8 int A_is_active (A *a);
9
10 float A_value (A *a);
11
12 void A_deactivate (A *a);
```

Listing 2421: system_a.c

```
1 #include "system_a.h"
2
3 void A_activate (A *a)
4 {
5     int i;
6
7     for (i = 0; i < 2; i++)
8     {
9         a->val[i] = 0.0;
10    }
11    a->active = 1;
12 }
13
14 int A_is_active (A *a)
15 {
16     return a->active == 1;
17 }
18
19 float A_value (A *a)
20 {
21     return (a->val[0] + a->val[1]) / 2.0;
22 }
23
24 void A_deactivate (A *a)
25 {
```

(continues on next page)

(continued from previous page)

```

26     a->active = 0;
27 }

```

Listing 2422: system_b.h

```

1  typedef struct {
2      float val;
3      int  active;
4  } B;
5
6  void B_activate (B *b);
7
8  int B_is_active (B *b);
9
10 float B_value (B *b);
11
12 void B_deactivate (B *b);

```

Listing 2423: system_b.c

```

1  #include "system_b.h"
2
3  void B_activate (B *b)
4  {
5      b->val    = 0.0;
6      b->active = 1;
7  }
8
9  int B_is_active (B *b)
10 {
11     return b->active == 1;
12 }
13
14 float B_value (B *b)
15 {
16     return b->val;
17 }
18
19 void B_deactivate (B *b)
20 {
21     b->active = 0;
22 }

```

Listing 2424: system_ab.h

```

1  #include "system_a.h"
2  #include "system_b.h"
3
4  typedef struct {
5      A a;
6      B b;
7  } AB;
8
9  void AB_activate (AB *ab);
10
11 int AB_is_active (AB *ab);
12
13 float AB_value (AB *ab);
14
15 int AB_check (AB *ab);

```

(continues on next page)

(continued from previous page)

```
16
17 void AB_deactivate (AB *ab);
```

Listing 2425: system_ab.c

```
1  #include <math.h>
2  #include "system_ab.h"
3
4  void AB_activate (AB *ab)
5  {
6      A_activate (&ab->a);
7      B_activate (&ab->b);
8  }
9
10 int AB_is_active (AB *ab)
11 {
12     return A_is_active(&ab->a) && B_is_active(&ab->b);
13 }
14
15 float AB_value (AB *ab)
16 {
17     return (A_value (&ab->a) + B_value (&ab->b)) / 2;
18 }
19
20 int AB_check (AB *ab)
21 {
22     const float threshold = 0.1;
23
24     return fabs (A_value (&ab->a) - B_value (&ab->b)) < threshold;
25 }
26
27 void AB_deactivate (AB *ab)
28 {
29     A_deactivate (&ab->a);
30     B_deactivate (&ab->b);
31 }
```

Listing 2426: main.c

```
1  #include <stdio.h>
2  #include "system_ab.h"
3
4  void display_active (AB *ab)
5  {
6      if (AB_is_active (ab))
7          printf ("System AB is active.\n");
8      else
9          printf ("System AB is not active.\n");
10 }
11
12 void display_check (AB *ab)
13 {
14     if (AB_check (ab))
15         printf ("System AB check: PASSED.\n");
16     else
17         printf ("System AB check: FAILED.\n");
18 }
19
20 int main()
21 {
```

(continues on next page)

(continued from previous page)

```

22     AB s;
23
24     printf ("Activating system AB...\n");
25     AB_activate (&s);
26
27     display_active (&s);
28     display_check (&s);
29
30     printf ("Deactivating system AB...\n");
31     AB_deactivate (&s);
32
33     display_active (&s);
34 }

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOn00P.System_AB_C
 MD5: 649bcfe39504c853a0c3f43e1e048f34

Runtime output

```

Activating system AB...
System AB is active.
System AB check: PASSED.
Deactivating system AB...
System AB is not active.

```

Here, each system is implemented in a separate set of header and source-code files. For example, the API of system AB is in `system_ab.h` and its implementation in `system_ab.c`.

In the main application, we instantiate system AB and activate it. Then, we proceed to display the activation state and the result of the system's health check. Finally, we deactivate the system and display the activation state again.

91.2.2 Initial translation to Ada

The direct implementation in Ada is:

[Ada]

Listing 2427: `system_a.ads`

```

1  package System_A is
2
3      type Val_Array is array (Positive range <>) of Float;
4
5      type A is record
6          Val      : Val_Array (1 .. 2);
7          Active   : Boolean;
8      end record;
9
10     procedure A_Activate (E : in out A);
11
12     function A_Is_Active (E : A) return Boolean;
13
14     function A_Value (E : A) return Float;
15
16     procedure A_Deactivate (E : in out A);
17
18 end System_A;

```

Listing 2428: system_a.adb

```
1 package body System_A is
2
3     procedure A_Activate (E : in out A) is
4     begin
5         E.Val      := (others => 0.0);
6         E.Active := True;
7     end A_Activate;
8
9     function A_Is_Active (E : A) return Boolean is
10    begin
11        return E.Active;
12    end A_Is_Active;
13
14    function A_Value (E : A) return Float is
15    begin
16        return (E.Val (1) + E.Val (2)) / 2.0;
17    end A_Value;
18
19    procedure A_Deactivate (E : in out A) is
20    begin
21        E.Active := False;
22    end A_Deactivate;
23
24 end System_A;
```

Listing 2429: system_b.ads

```
1 package System_B is
2
3     type B is record
4         Val      : Float;
5         Active   : Boolean;
6     end record;
7
8     procedure B_Activate (E : in out B);
9
10    function B_Is_Active (E : B) return Boolean;
11
12    function B_Value (E : B) return Float;
13
14    procedure B_Deactivate (E : in out B);
15
16 end System_B;
```

Listing 2430: system_b.adb

```
1 package body System_B is
2
3     procedure B_Activate (E : in out B) is
4     begin
5         E.Val      := 0.0;
6         E.Active := True;
7     end B_Activate;
8
9     function B_Is_Active (E : B) return Boolean is
10    begin
11        return E.Active;
12    end B_Is_Active;
13
```

(continues on next page)

(continued from previous page)

```

14  function B_Value (E : B) return Float is
15  begin
16      return E.Val;
17  end B_Value;
18
19  procedure B_Deactivate (E : in out B) is
20  begin
21      E.Active := False;
22  end B_Deactivate;
23
24  end System_B;

```

Listing 2431: system_ab.ads

```

1  with System_A; use System_A;
2  with System_B; use System_B;
3
4  package System_AB is
5
6      type AB is record
7          SA : A;
8          SB : B;
9      end record;
10
11     procedure AB_Activate (E : in out AB);
12
13     function AB_Is_Active (E : AB) return Boolean;
14
15     function AB_Value (E : AB) return Float;
16
17     function AB_Check (E : AB) return Boolean;
18
19     procedure AB_Deactivate (E : in out AB);
20
21 end System_AB;

```

Listing 2432: system_ab.adb

```

1  package body System_AB is
2
3      procedure AB_Activate (E : in out AB) is
4      begin
5          A_Activate (E.SA);
6          B_Activate (E.SB);
7      end AB_Activate;
8
9      function AB_Is_Active (E : AB) return Boolean is
10     begin
11         return A_Is_Active (E.SA) and B_Is_Active (E.SB);
12     end AB_Is_Active;
13
14     function AB_Value (E : AB) return Float is
15     begin
16         return (A_Value (E.SA) + B_Value (E.SB)) / 2.0;
17     end AB_Value;
18
19     function AB_Check (E : AB) return Boolean is
20     Threshold : constant := 0.1;
21     begin
22         return abs (A_Value (E.SA) - B_Value (E.SB)) < Threshold;

```

(continues on next page)

(continued from previous page)

```
23     end AB_Check;
24
25     procedure AB_Deactivate (E : in out AB) is
26     begin
27         A_Deactivate (E.SA);
28         B_Deactivate (E.SB);
29     end AB_Deactivate;
30
31 end System_AB;
```

Listing 2433: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with System_AB;   use System_AB;
4
5  procedure Main is
6
7      procedure Display_Active (E : AB) is
8      begin
9          if AB_Is_Active (E) then
10             Put_Line ("System AB is active");
11          else
12             Put_Line ("System AB is not active");
13          end if;
14      end Display_Active;
15
16      procedure Display_Check (E : AB) is
17      begin
18          if AB_Check (E) then
19             Put_Line ("System AB check: PASSED");
20          else
21             Put_Line ("System AB check: FAILED");
22          end if;
23      end Display_Check;
24
25      S : AB;
26  begin
27      Put_Line ("Activating system AB...");
28      AB_Activate (S);
29
30      Display_Active (S);
31      Display_Check (S);
32
33      Put_Line ("Deactivating system AB...");
34      AB_Deactivate (S);
35
36      Display_Active (S);
37  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOn00P.System_AB_Ada
MD5: f2e3df0b3874e5edc5ea90c01961cf64

Runtime output

```
Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
```

(continues on next page)

(continued from previous page)

System AB is not active

As you can see, this is a direct translation that doesn't change much of the structure of the original C code. Here, the goal was to simply translate the system from one language to another and make sure that the behavior remains the same.

91.2.3 Improved Ada implementation

By analyzing this direct implementation, we may notice the following points:

- Packages `System_A`, `System_B` and `System_AB` are used to describe aspects of the same system. Instead of having three distinct packages, we could group them as child packages of a common parent package — let's call it `Simple`, since this system is supposed to be simple. This approach has the advantage of allowing us to later use the parent package to implement functionality that is common for all parts of the system.
- Since we have subprograms that operate on types `A`, `B` and `AB`, we should avoid exposing the record components by moving the type declarations to the private part of the corresponding packages.
- Since Ada supports subprogram overloading — as discussed in [this section from chapter 2](#) (page 1802) —, we don't need to have different names for subprograms with similar functionality. For example, instead of having `A_Is_Active` and `B_Is_Active`, we can simply name these functions `Is_Active` for both types `A` and `B`.
- Some of the functions — such as `A_Is_Active` and `A_Value` — are very simple, so we could simplify them with expression functions.

This is an update to the implementation that addresses all the points above:

[Ada]

Listing 2434: `simple.ads`

```

1 package Simple
2   with Pure
3 is
4 end Simple;
```

Listing 2435: `simple-system_a.ads`

```

1 package Simple.System_A is
2
3   type A is private;
4
5   procedure Activate (E : in out A);
6
7   function Is_Active (E : A) return Boolean;
8
9   function Value (E : A) return Float;
10
11  procedure Finalize (E : in out A);
12
13 private
14
15  type Val_Array is array (Positive range <>) of Float;
16
17  type A is record
18    Val    : Val_Array (1 .. 2);
19    Active : Boolean;
```

(continues on next page)

(continued from previous page)

```
20     end record;  
21  
22 end Simple.System_A;
```

Listing 2436: simple-system_a.adb

```
1 package body Simple.System_A is  
2  
3     procedure Activate (E : in out A) is  
4     begin  
5         E.Val := (others => 0.0);  
6         E.Active := True;  
7     end Activate;  
8  
9     function Is_Active (E : A) return Boolean is  
10         (E.Active);  
11  
12     function Value (E : A) return Float is  
13     begin  
14         return (E.Val (1) + E.Val (2)) / 2.0;  
15     end Value;  
16  
17     procedure Finalize (E : in out A) is  
18     begin  
19         E.Active := False;  
20     end Finalize;  
21  
22 end Simple.System_A;
```

Listing 2437: simple-system_b.ads

```
1 package Simple.System_B is  
2  
3     type B is private;  
4  
5     procedure Activate (E : in out B);  
6  
7     function Is_Active (E : B) return Boolean;  
8  
9     function Value (E : B) return Float;  
10  
11     procedure Finalize (E : in out B);  
12  
13 private  
14  
15     type B is record  
16         Val : Float;  
17         Active : Boolean;  
18     end record;  
19  
20 end Simple.System_B;
```

Listing 2438: simple-system_b.adb

```
1 package body Simple.System_B is  
2  
3     procedure Activate (E : in out B) is  
4     begin  
5         E.Val := 0.0;  
6         E.Active := True;
```

(continues on next page)

(continued from previous page)

```

7   end Activate;
8
9   function Is_Active (E : B) return Boolean is
10  begin
11      return E.Active;
12  end Is_Active;
13
14  function Value (E : B) return Float is
15      (E.Val);
16
17  procedure Finalize (E : in out B) is
18  begin
19      E.Active := False;
20  end Finalize;
21
22  end Simple.System_B;

```

Listing 2439: simple-system_ab.ads

```

1  with Simple.System_A; use Simple.System_A;
2  with Simple.System_B; use Simple.System_B;
3
4  package Simple.System_AB is
5
6      type AB is private;
7
8      procedure Activate (E : in out AB);
9
10     function Is_Active (E : AB) return Boolean;
11
12     function Value (E : AB) return Float;
13
14     function Check (E : AB) return Boolean;
15
16     procedure Finalize (E : in out AB);
17
18  private
19
20     type AB is record
21         SA : A;
22         SB : B;
23     end record;
24
25  end Simple.System_AB;

```

Listing 2440: simple-system_ab.adb

```

1  package body Simple.System_AB is
2
3      procedure Activate (E : in out AB) is
4      begin
5          Activate (E.SA);
6          Activate (E.SB);
7      end Activate;
8
9      function Is_Active (E : AB) return Boolean is
10         (Is_Active (E.SA) and Is_Active (E.SB));
11
12     function Value (E : AB) return Float is
13         ((Value (E.SA) + Value (E.SB)) / 2.0);

```

(continues on next page)

(continued from previous page)

```
14
15  function Check (E : AB) return Boolean is
16      Threshold : constant := 0.1;
17  begin
18      return abs (Value (E.SA) - Value (E.SB)) < Threshold;
19  end Check;
20
21  procedure Finalize (E : in out AB) is
22  begin
23      Finalize (E.SA);
24      Finalize (E.SB);
25  end Finalize;
26
27  end Simple.System_AB;
```

Listing 2441: main.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Simple.System_AB; use Simple.System_AB;
4
5  procedure Main is
6
7      procedure Display_Active (E : AB) is
8      begin
9          if Is_Active (E) then
10             Put_Line ("System AB is active");
11         else
12             Put_Line ("System AB is not active");
13         end if;
14     end Display_Active;
15
16     procedure Display_Check (E : AB) is
17     begin
18         if Check (E) then
19             Put_Line ("System AB check: PASSED");
20         else
21             Put_Line ("System AB check: FAILED");
22         end if;
23     end Display_Check;
24
25     S : AB;
26  begin
27     Put_Line ("Activating system AB...");
28     Activate (S);
29
30     Display_Active (S);
31     Display_Check (S);
32
33     Put_Line ("Deactivating system AB...");
34     Finalize (S);
35
36     Display_Active (S);
37  end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOn00P.System_AB_Ada_Enhanced
MD5: 5019a7088ab4160f5e3b33c73db2b03b

Runtime output

```

Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active

```

91.3 First Object-Oriented Approach

Until now, we haven't used any of the object-oriented programming features of the Ada language. So we can start by analyzing the API of systems A and B and deciding how to best abstract some of its elements using object-oriented programming.

91.3.1 Interfaces

The first thing we may notice is that we actually have two distinct sets of APIs there:

- one API for activating and deactivating the system.
- one API for retrieving the value of the system.

We can use this distinction to declare two interface types:

- `Activation_IF` for the `Activate` and `Deactivate` procedures and the `Is_Active` function;
- `Value_Retrieval_IF` for the `Value` function.

This is how the declaration could look like:

```

type Activation_IF is interface;

procedure Activate (E : in out Activation_IF) is abstract;
function Is_Active (E : Activation_IF) return Boolean is abstract;
procedure Deactivate (E : in out Activation_IF) is abstract;

type Value_Retrieval_IF is interface;

function Value (E : Value_Retrieval_IF) return Float is abstract;

```

Note that, because we are declaring interface types, all operations on those types must be abstract or, in the case of procedures, they can also be declared `null`. For example, we could change the declaration of the procedures above to this:

```

procedure Activate (E : in out Activation_IF) is null;
procedure Deactivate (E : in out Activation_IF) is null;

```

When an operation is declared abstract, we must override it for the type that derives from the interface. When a procedure is declared `null`, it acts as a do-nothing default. In this case, overriding the operation is optional for the type that derives from this interface.

91.3.2 Base type

Since the original system needs both interfaces we've just described, we have to declare another type that combines those interfaces. We can do this by declaring the interface type `Sys_Base`, which serves as the base type for systems A and B. This is the declaration:

```

type Sys_Base is interface and Activation_IF and Value_Retrieval_IF;

```

Since the system activation functionality is common for both systems A and B, we could implement it as part of Sys_Base. That would require changing the declaration from a simple interface to an abstract record:

```
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
  with null record;
```

Now, we can add the Boolean component to the record (as a private component) and override the subprograms of the Activation_IF interface. This is the adapted declaration:

```
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with private;

overriding procedure Activate (E : in out Sys_Base);
overriding function Is_Active (E : Sys_Base) return Boolean;
overriding procedure Deactivate (E : in out Sys_Base);

private

type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with record
  Active : Boolean;
end record;
```

91.3.3 Derived types

In the declaration of the Sys_Base type we've just seen, we're not overriding the Value function — from the Value_Retrieval_IF interface — for the Sys_Base type, so it remains an abstract function for Sys_Base. Therefore, the Sys_Base type itself remains abstract and needs to be explicitly declared as such.

We use this strategy to ensure that all types derived from Sys_Base need to implement their own version of the Value function. For example:

```
type A is new Sys_Base with private;

overriding function Value (E : A) return Float;
```

Here, the A type is derived from the Sys_Base and it includes its own version of the Value function by overriding it. Therefore, A is not an abstract type anymore and can be used to declare objects:

```
procedure Main is
  Obj : A;
  V : Float;
begin
  Obj.Activate;
  V := Obj.Value;
end Main;
```

Important

Note that the use of the **overriding** keyword in the subprogram declaration is not strictly necessary. In fact, we could leave this keyword out, and the code would still compile. However, if provided, the compiler will check whether the information is correct.

Using the **overriding** keyword can help to avoid bad surprises — when you *may think* that you're overriding a subprogram, but you're actually not. Similarly, you can also write **not overriding** to be explicit about subprograms that are new primitives of a derived type. For example:

```
not overriding function Check (E : AB) return Boolean;
```

We also need to declare the values that are used internally in systems A and B. For system A, this is the declaration:

```
type A is new Sys_Base with private;

overriding function Value (E : A) return Float;

private

type Val_Array is array (Positive range <>) of Float;

type A is new Sys_Base with record
    Val : Val_Array (1 .. 2);
end record;
```

91.3.4 Subprograms from parent

In the previous implementation, we've seen that the A_Activate and B_Activate procedures perform the following steps:

- initialize internal values;
- indicate that the system is active (by setting the Active flag to **True**).

In the implementation of the Activate procedure for the Sys_Base type, however, we're only dealing with the second step. Therefore, we need to override the Activate procedure and make sure that we initialize internal values as well. First, we need to declare this procedure for type A:

```
type A is new Sys_Base with private;

overriding procedure Activate (E : in out A);
```

In the implementation of Activate, we should call the Activate procedure from the parent (Sys_Base) to ensure that whatever was performed for the parent will be performed in the derived type as well. For example:

```
overriding procedure Activate (E : in out A) is
begin
    E.Val := (others => 0.0);
    Sys_Base (E).Activate;    -- Calling Activate for Sys_Base type:
                             -- this call initializes the Active flag.
end;
```

Here, by writing Sys_Base (E), we're performing a view conversion. Basically, we're telling the compiler to view E not as an object of type A, but of type Sys_Base. When we do this, any operation performed on this object will be done as if it was an object of Sys_Base type, which includes calling the Activate procedure of the Sys_Base type.

Important

If we write `T (Obj).Proc`, we're telling the compiler to call the Proc procedure of type T and apply it on Obj.

If we write `T'Class (Obj).Proc`, however, we're telling the compiler to dispatch the call. For example, if Obj is of derived type T2 and there's an overridden Proc procedure for type T2, then this procedure will be called instead of the Proc procedure for type T.

91.3.5 Type AB

While the implementation of systems A and B is almost straightforward, it gets more interesting in the case of system AB. Here, we have a similar API, but we don't need the activation mechanism implemented in the abstract type `Sys_Base`. Therefore, deriving from `Sys_Base` is not the best option. Instead, when declaring the AB type, we can simply use the same interfaces as we did for `Sys_Base`, but keep it independent from `Sys_Base`. For example:

```
type AB is new Activation_IF and Value_Retrieval_IF with private;

private

type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
```

Naturally, we still need to override all the subprograms that are part of the `Activation_IF` and `Value_Retrieval_IF` interfaces. Also, we need to implement the additional `Check` function that was originally only available on system AB. Therefore, we declare these subprograms:

```
overriding procedure Activate (E : in out AB);
overriding function Is_Active (E : AB) return Boolean;
overriding procedure Deactivate (E : in out AB);

overriding function Value (E : AB) return Float;

not overriding function Check (E : AB) return Boolean;
```

91.3.6 Updated source-code

Finally, this is the complete source-code example:

[Ada]

Listing 2442: simple.ads

```
1 package Simple is
2
3   type Activation_IF is interface;
4
5   procedure Activate (E : in out Activation_IF) is abstract;
6   function Is_Active (E : Activation_IF) return Boolean is abstract;
7   procedure Deactivate (E : in out Activation_IF) is abstract;
8
9   type Value_Retrieval_IF is interface;
10
11   function Value (E : Value_Retrieval_IF) return Float is abstract;
12
13   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
14     with private;
15
16   overriding procedure Activate (E : in out Sys_Base);
17   overriding function Is_Active (E : Sys_Base) return Boolean;
18   overriding procedure Deactivate (E : in out Sys_Base);
19
20 private
21
22   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
```

(continues on next page)

(continued from previous page)

```

23     with record
24         Active : Boolean;
25     end record;
26
27 end Simple;

```

Listing 2443: simple.adb

```

1  package body Simple is
2
3      overriding procedure Activate (E : in out Sys_Base) is
4      begin
5          E.Active := True;
6      end Activate;
7
8      overriding function Is_Active (E : Sys_Base) return Boolean is
9          (E.Active);
10
11     overriding procedure Deactivate (E : in out Sys_Base) is
12     begin
13         E.Active := False;
14     end Deactivate;
15
16 end Simple;

```

Listing 2444: simple-system_a.ads

```

1  package Simple.System_A is
2
3      type A is new Sys_Base with private;
4
5      overriding procedure Activate (E : in out A);
6
7      overriding function Value (E : A) return Float;
8
9  private
10
11     type Val_Array is array (Positive range <>) of Float;
12
13     type A is new Sys_Base with record
14         Val : Val_Array (1 .. 2);
15     end record;
16
17 end Simple.System_A;

```

Listing 2445: simple-system_a.adb

```

1  package body Simple.System_A is
2
3      procedure Activate (E : in out A) is
4      begin
5          E.Val := (others => 0.0);
6          Sys_Base (E).Activate;
7      end Activate;
8
9      function Value (E : A) return Float is
10         pragma Assert (E.Val'Length = 2);
11     begin
12         return (E.Val (1) + E.Val (2)) / 2.0;
13     end Value;

```

(continues on next page)

(continued from previous page)

```
14  
15 end Simple.System_A;
```

Listing 2446: simple-system_b.ads

```
1 package Simple.System_B is  
2  
3   type B is new Sys_Base with private;  
4  
5   overriding procedure Activate (E : in out B);  
6  
7   overriding function Value (E : B) return Float;  
8  
9 private  
10  
11   type B is new Sys_Base with record  
12     Val : Float;  
13   end record;  
14  
15 end Simple.System_B;
```

Listing 2447: simple-system_b.adb

```
1 package body Simple.System_B is  
2  
3   procedure Activate (E : in out B) is  
4   begin  
5     E.Val := 0.0;  
6     Sys_Base (E).Activate;  
7   end Activate;  
8  
9   function Value (E : B) return Float is  
10     (E.Val);  
11  
12 end Simple.System_B;
```

Listing 2448: simple-system_ab.ads

```
1 with Simple.System_A; use Simple.System_A;  
2 with Simple.System_B; use Simple.System_B;  
3  
4 package Simple.System_AB is  
5  
6   type AB is new Activation_IF and Value_Retrieval_IF with private;  
7  
8   overriding procedure Activate (E : in out AB);  
9   overriding function Is_Active (E : AB) return Boolean;  
10  overriding procedure Deactivate (E : in out AB);  
11  
12  overriding function Value (E : AB) return Float;  
13  
14  not overriding function Check (E : AB) return Boolean;  
15  
16 private  
17  
18  type AB is new Activation_IF and Value_Retrieval_IF with record  
19    SA : A;  
20    SB : B;  
21  end record;  
22
```

(continues on next page)

(continued from previous page)

```
23 end Simple.System_AB;
```

Listing 2449: simple-system_ab.adb

```
1 package body Simple.System_AB is
2
3   procedure Activate (E : in out AB) is
4   begin
5     E.SA.Activate;
6     E.SB.Activate;
7   end Activate;
8
9   function Is_Active (E : AB) return Boolean is
10    (E.SA.Is_Active and E.SB.Is_Active);
11
12   procedure Deactivate (E : in out AB) is
13   begin
14     E.SA.Deactivate;
15     E.SB.Deactivate;
16   end Deactivate;
17
18   function Value (E : AB) return Float is
19    ((E.SA.Value + E.SB.Value) / 2.0);
20
21   function Check (E : AB) return Boolean is
22    Threshold : constant := 0.1;
23   begin
24     return abs (E.SA.Value - E.SB.Value) < Threshold;
25   end Check;
26
27 end Simple.System_AB;
```

Listing 2450: main.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Simple.System_AB; use Simple.System_AB;
4
5 procedure Main is
6
7   procedure Display_Active (E : AB) is
8   begin
9     if Is_Active (E) then
10      Put_Line ("System AB is active");
11     else
12      Put_Line ("System AB is not active");
13     end if;
14   end Display_Active;
15
16   procedure Display_Check (E : AB) is
17   begin
18     if Check (E) then
19      Put_Line ("System AB check: PASSED");
20     else
21      Put_Line ("System AB check: FAILED");
22     end if;
23   end Display_Check;
24
25   S : AB;
26 begin
```

(continues on next page)

(continued from previous page)

```
27   Put_Line ("Activating system AB...");
28   Activate (S);
29
30   Display_Active (S);
31   Display_Check (S);
32
33   Put_Line ("Deactivating system AB...");
34   Deactivate (S);
35
36   Display_Active (S);
37 end Main;
```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOnOOP.System_AB_Ada_OOP_1
MD5: 02adee1f81b025007244bd6d13e8b5a3

Runtime output

```
Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active
```

91.4 Further Improvements

When analyzing the complete source-code, we see that there are at least two areas that we could still improve.

91.4.1 Dispatching calls

The first issue concerns the implementation of the Activate procedure for types derived from Sys_Base. For those derived types, we're expecting that the Activate procedure of the parent must be called in the implementation of the overriding Activate procedure. For example:

```
package body Simple.System_A is

  procedure Activate (E : in out A) is
  begin
    E.Val := (others => 0.0);
    Activate (Sys_Base (E));
  end;
```

If a developer forgets to call that specific Activate procedure, however, the system won't work as expected. A better strategy could be the following:

- Declare a new Activation_Reset procedure for Sys_Base type.
- Make a dispatching call to the Activation_Reset procedure in the body of the Activate procedure (of the Sys_Base type).
- Let the derived types implement their own version of the Activation_Reset procedure.

This is a simplified view of the implementation using the points described above:

```

package Simple is

    type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
        private;

    not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;

end Simple;

package body Simple is

    procedure Activate (E : in out Sys_Base) is
    begin
        -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
        --         We need to use the 'Class attribute here --- not using this
        --         attribute is an error that will be caught by the compiler.
        Sys_Base'Class (E).Activation_Reset;

        E.Active := True;
    end Activate;

end Simple;

package Simple.System_A is

    type A is new Sys_Base with private;

private

    type Val_Array is array (Positive range <>) of Float;

    type A is new Sys_Base with record
        Val : Val_Array (1 .. 2);
    end record;

    overriding procedure Activation_Reset (E : in out A);

end Simple.System_A;

package body Simple.System_A is

    procedure Activation_Reset (E : in out A) is
    begin
        E.Val := (others => 0.0);
    end Activation_Reset;

end Simple.System_A;

```

An important detail is that, in the implementation of `Activate`, we use `Sys_Base'Class` to ensure that the call to `Activation_Reset` will dispatch. If we had just written `E.Activation_Reset` instead, then we would be calling the `Activation_Reset` procedure of `Sys_Base` itself, which is not what we actually want here. The compiler will catch the error if you don't do the conversion to the class-wide type, because it would otherwise be a statically-bound call to an abstract procedure, which is illegal at compile-time.

91.4.2 Dynamic allocation

The next area that we could improve is in the declaration of the system AB. In the previous implementation, we were explicitly describing the two components of that system, namely a component of type A and a component of type B:

```
type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
```

Of course, this declaration matches the system requirements that we presented in the beginning. However, we could use strategies that make it easier to incorporate requirement changes later on. For example, we could hide this information about systems A and B by simply declaring an array of components of type `access Sys_Base'Class` and allocate them dynamically in the body of the package. Naturally, this approach might not be suitable for certain platforms. However, the advantage would be that, if we wanted to replace the component of type B by a new component of type C, for example, we wouldn't need to change the interface. This is how the updated declaration could look like:

```
type Sys_Base_Class_Access is access Sys_Base'Class;
type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;

type AB is limited new Activation_IF and Value_Retrieval_IF with record
  S_Array : Sys_Base_Array (1 .. 2);
end record;
```

Important

Note that we're now using the `limited` keyword in the declaration of type AB. That is necessary because we want to prevent objects of type AB being copied by assignment, which would lead to two objects having the same (dynamically allocated) subsystems A and B internally. This change requires that both `Activation_IF` and `Value_Retrieval_IF` are declared limited as well.

The body of `Activate` could then allocate those components:

```
procedure Activate (E : in out AB) is
begin
  E.S_Array := (new A, new B);
  for S of E.S_Array loop
    S.Activate;
  end loop;
end Activate;
```

And the body of `Deactivate` could deallocate them:

```
procedure Deactivate (E : in out AB) is
  procedure Free is
    new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
  begin
    for S of E.S_Array loop
      S.Deactivate;
      Free (S);
    end loop;
  end Deactivate;
```

91.4.3 Limited controlled types

Another approach that we could use to implement the dynamic allocation of systems A and B is to declare AB as a limited controlled type — based on the `Limited_Controlled` type of the `Ada.Finalization` package.

The `Limited_Controlled` type includes the following operations:

- Initialize, which is called when objects of a type derived from the Limited_Controlled type are being created — by declaring an object of the derived type, for example —, and
- Finalize, which is called when objects are being destroyed — for example, when an object gets out of scope at the end of a subprogram where it was created.

In this case, we must override those procedures, so we can use them for dynamic memory allocation. This is a simplified view of the update implementation:

```
package Simple.System_AB is

  type AB is limited new Ada.Finalization.Limited_Controlled and
    Activation_IF and Value_Retrieval_IF with private;

  overriding procedure Initialize (E : in out AB);
  overriding procedure Finalize   (E : in out AB);

end Simple.System_AB;

package body Simple.System_AB is

  overriding procedure Initialize (E : in out AB) is
  begin
    E.S_Array := (new A, new B);
  end Initialize;

  overriding procedure Finalize   (E : in out AB) is
    procedure Free is
      new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
    begin
      for S of E.S_Array loop
        Free (S);
      end loop;
    end Finalize;

end Simple.System_AB;
```

91.4.4 Updated source-code

Finally, this is the complete updated source-code example:

[Ada]

Listing 2451: simple.ads

```
1 package Simple is
2
3   type Activation_IF is limited interface;
4
5   procedure Activate (E : in out Activation_IF) is abstract;
6   function Is_Active (E : Activation_IF) return Boolean is abstract;
7   procedure Deactivate (E : in out Activation_IF) is abstract;
8
9   type Value_Retrieval_IF is limited interface;
10
11   function Value (E : Value_Retrieval_IF) return Float is abstract;
12
13   type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
14     private;
15
16   overriding procedure Activate (E : in out Sys_Base);
```

(continues on next page)

(continued from previous page)

```
17 overriding function Is_Active (E : Sys_Base) return Boolean;
18 overriding procedure Deactivate (E : in out Sys_Base);
19
20 not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;
21
22 private
23
24 type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
25     record
26         Active : Boolean;
27     end record;
28
29 end Simple;
```

Listing 2452: simple.adb

```
1 package body Simple is
2
3     procedure Activate (E : in out Sys_Base) is
4     begin
5         -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
6         -- We need to use the 'Class attribute:
7         Sys_Base'Class (E).Activation_Reset;
8
9         E.Active := True;
10    end Activate;
11
12    function Is_Active (E : Sys_Base) return Boolean is
13    (E.Active);
14
15    procedure Deactivate (E : in out Sys_Base) is
16    begin
17        E.Active := False;
18    end Deactivate;
19
20 end Simple;
```

Listing 2453: simple-system_a.ads

```
1 package Simple.System_A is
2
3     type A is new Sys_Base with private;
4
5     overriding function Value (E : A) return Float;
6
7 private
8
9     type Val_Array is array (Positive range <>) of Float;
10
11     type A is new Sys_Base with record
12         Val : Val_Array (1 .. 2);
13     end record;
14
15     overriding procedure Activation_Reset (E : in out A);
16
17 end Simple.System_A;
```

Listing 2454: simple-system_a.adb

```

1 package body Simple.System_A is
2
3     procedure Activation_Reset (E : in out A) is
4     begin
5         E.Val := (others => 0.0);
6     end Activation_Reset;
7
8     function Value (E : A) return Float is
9     pragma Assert (E.Val'Length = 2);
10    begin
11        return (E.Val (1) + E.Val (2)) / 2.0;
12    end Value;
13
14 end Simple.System_A;
```

Listing 2455: simple-system_b.ads

```

1 package Simple.System_B is
2
3     type B is new Sys_Base with private;
4
5     overriding function Value (E : B) return Float;
6
7 private
8
9     type B is new Sys_Base with record
10         Val : Float;
11     end record;
12
13     overriding procedure Activation_Reset (E : in out B);
14
15 end Simple.System_B;
```

Listing 2456: simple-system_b.adb

```

1 package body Simple.System_B is
2
3     procedure Activation_Reset (E : in out B) is
4     begin
5         E.Val := 0.0;
6     end Activation_Reset;
7
8     function Value (E : B) return Float is
9     (E.Val);
10
11 end Simple.System_B;
```

Listing 2457: simple-system_ab.ads

```

1 with Ada.Finalization;
2
3 package Simple.System_AB is
4
5     type AB is limited new Ada.Finalization.Limited_Controlled and
6         Activation_IF and Value_Retrieval_IF with private;
7
8     overriding procedure Activate (E : in out AB);
9     overriding function Is_Active (E : AB) return Boolean;
```

(continues on next page)

(continued from previous page)

```

10  overriding procedure Deactivate (E : in out AB);
11
12  overriding function Value (E : AB) return Float;
13
14  not overriding function Check (E : AB) return Boolean;
15
16  private
17
18  type Sys_Base_Class_Access is access Sys_Base'Class;
19  type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;
20
21  type AB is limited new Ada.Finalization.Limited_Controlled and
22    Activation_IF and Value_Retrieval_IF with record
23    S_Array : Sys_Base_Array (1 .. 2);
24  end record;
25
26  overriding procedure Initialize (E : in out AB);
27  overriding procedure Finalize   (E : in out AB);
28
29  end Simple.System_AB;

```

Listing 2458: simple-system_ab.adb

```

1  with Ada.Unchecked_Deallocation;
2
3  with Simple.System_A; use Simple.System_A;
4  with Simple.System_B; use Simple.System_B;
5
6  package body Simple.System_AB is
7
8    overriding procedure Initialize (E : in out AB) is
9    begin
10      E.S_Array := (new A, new B);
11    end Initialize;
12
13    overriding procedure Finalize   (E : in out AB) is
14      procedure Free is
15        new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
16      begin
17        for S of E.S_Array loop
18          Free (S);
19        end loop;
20      end Finalize;
21
22      procedure Activate (E : in out AB) is
23      begin
24        for S of E.S_Array loop
25          S.Activate;
26        end loop;
27      end Activate;
28
29      function Is_Active (E : AB) return Boolean is
30        (for all S of E.S_Array => S.Is_Active);
31
32      procedure Deactivate (E : in out AB) is
33      begin
34        for S of E.S_Array loop
35          S.Deactivate;
36        end loop;
37      end Deactivate;
38

```

(continues on next page)

(continued from previous page)

```

39  function Value (E : AB) return Float is
40      ((E.S_Array (1).Value + E.S_Array (2).Value) / 2.0);
41
42  function Check (E : AB) return Boolean is
43      Threshold : constant := 0.1;
44  begin
45      return abs (E.S_Array (1).Value - E.S_Array (2).Value) < Threshold;
46  end Check;
47
48  end Simple.System_AB;

```

Listing 2459: main.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Simple.System_AB; use Simple.System_AB;
4
5  procedure Main is
6
7      procedure Display_Active (E : AB) is
8      begin
9          if Is_Active (E) then
10             Put_Line ("System AB is active");
11          else
12             Put_Line ("System AB is not active");
13          end if;
14      end Display_Active;
15
16      procedure Display_Check (E : AB) is
17      begin
18          if Check (E) then
19             Put_Line ("System AB check: PASSED");
20          else
21             Put_Line ("System AB check: FAILED");
22          end if;
23      end Display_Check;
24
25      S : AB;
26  begin
27      Put_Line ("Activating system AB...");
28      Activate (S);
29
30      Display_Active (S);
31      Display_Check (S);
32
33      Put_Line ("Deactivating system AB...");
34      Deactivate (S);
35
36      Display_Active (S);
37  end Main;

```

Code block metadata

Project: Courses.Ada_For_Embedded_C_Dev.HandsOnOOP.System_AB_Ada_OOP_2
MD5: f8d0d4a07aaa045cb30bddc88db2215a

Runtime output

```

Activating system AB...
System AB is active
System AB check: PASSED

```

(continues on next page)

(continued from previous page)

```
Deactivating system AB...  
System AB is not active
```

Naturally, this is by no means the best possible implementation of system AB. By applying other software design strategies that we haven't covered here, we could most probably think of different ways to use object-oriented programming to improve this implementation. Also, in comparison to the *original implementation* (page 1969), we recognize that the amount of source-code has grown. On the other hand, we now have a system that is factored nicely, and also more extensible.

Part IX

Guidelines for Safe and Secure Ada/SPARK

Copyright © 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](http://creativecommons.org/licenses/by-sa/4.0)⁴⁴⁹

This document provides a reasonable set of coding standards to be applied to Ada/SPARK source code. The contents can be used as-is, or customized for a particular project.

This document was originally written by Patrick Rogers, and modified by Michael Frank.

⁴⁴⁹ <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

Ada is a general purpose, high-level programming language designed to support the construction of long-lived, highly-reliable applications. Like all general-purpose languages, only a subset of the full language is appropriate for safety-critical applications because the full language includes facilities that are difficult to analyze and verify to the degree required. This document facilitates identification of subsets appropriate for the highest levels of integrity, including safety-critical applications.

SPARK is a statically verifiable subset of Ada designed specifically for the most critical applications. Ada constructs not amenable to verification are precluded, such as arbitrary use of access types and full tasking. SPARK is also a superset of Ada, with additional contracts for specifying and verifying programs. Many of the guidelines (and more) are implicit in the design of SPARK.

Therefore, this document defines guidelines for the development of high-integrity, safety-critical applications in either the Ada or SPARK programming languages, or both (because the two can be mixed).

92.1 Scope

This document provides guidelines for development decisions, both at the system level and at the unit level, regarding the use of the programming languages Ada and SPARK, as well as related tools, such as static analyzers and unit test generators. It is not concerned with presentation issues such as naming, use of whitespace, or the like.

92.2 Structure

Rather than defining a specific set of rules defining a single subset, this document defines a set of criteria, in the form of guidelines, used by system architects to identify project-specific subsets appropriate to a given project.

The guidelines are separated into related categories, such as storage management, object-oriented programming, concurrency management, and so on. Each guideline is in a separate table, specifying the rule name, a unique identifier, and additional attributes common to each table.

92.3 Enforcement

Detection and enforcement mechanisms are indicated for each guideline. These mechanisms typically consist of the application of a language standard pragma named `Restrictions`, with policy-specific restriction identifiers given as parameters to the pragma [AdaRM2016]. Violations of the given restrictions are then detected and enforced by the Ada compiler.

Alternatively, the AdaCore GNATcheck utility program has rules precisely corresponding to those restriction identifiers, with the same degree of detection and enforcement. For example, the language restriction identifier `No_Unchecked_Deallocation` corresponds to the GNATcheck **+RRestrictions:No_Unchecked_Deallocation** rule.

The advantage of GNATcheck over the compiler is that all generated messages will be collected in the GNATcheck report that can be used as evidence of the level of adherence to the coding standard. In addition, GNATcheck provides a mechanism to deal with accepted exemptions. Note that, when the verification method indicates a GNATcheck rule could be used, the rule will note whether it is part of the standard GNATcheck rule set, or has been provided as-is within the document repository, located [here](#).⁴⁵⁰

In some cases the enforcement mechanism is the SPARK language and analyzer. Many of the guidelines (and more) are implicit in the design of SPARK and are, therefore, automatically enforced.

In some (very) rare cases the enforcement mechanism is manual program inspection, although alternatives (e.g., SPARK) are usually available and recommended. These guidelines are included because they are considered invaluable in this domain.

92.4 About the Rules

Although we refer to them as **rules** in the tables for the sake of brevity, these entries should be considered **guidance** because they require both thought and consideration of project-specific characteristics. For example, in some cases the guidance is to make a selection from among a set of distinct enumerated policies. In other cases a single guideline should be followed but not without some exceptional situations allowing it to be violated. The project lead should consider which guidelines to apply and how best to apply each guideline selected.

92.4.1 Mapping to Other Standards

Many of these rules can also be considered *good* programming practices. As such, many of them can be directly correlated to the *ISO/IEC Guidance to Avoiding Vulnerabilities in Programming Languages* [TR24772]. When a rule addresses one of these vulnerabilities, it is listed in the appropriate subsection.

In addition, MITRE's list of Common Weakness Enumerations [MITRE_CWE] contains many software issues that can be addressed by rules within this standard. Where appropriate, each rule lists the CWE(s) that can be addressed. Note that software CWEs tend to be generalized across all languages, so that many of the weaknesses may be prevented by the language itself. For this reason, the CWEs identified within this document specifically address vulnerabilities that would not be addressed by the Ada language itself (i.e., using the language is not sufficient to prevent the vulnerability).

⁴⁵⁰ https://github.com/AdaCore/learn/tree/master/content/courses/Guidelines_for_Safe_and_Secure_Ada_SPARK/rules

DEFINITIONS

This section contains terms and values used in the definitions of the rules set forth in this chapter.

93.1 Level

Level is the compliance level for the rule. Possible values are:

Mandatory

Non-compliance with a *Mandatory* recommendation level corresponds to a **high risk** of a software bug. There would need to be a good reason for non-conformity to a mandatory rule and, although it is accepted that exceptional cases may exist, any non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

Required

Non-compliance with a *Required* recommendation level corresponds to a **medium to high risk** of a software bug. Much like a *Mandatory* recommendation, there would need to be a good reason for non-conformity to a required rule. Although it is accepted that more exceptional cases may exist, non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

Advisory

Failure to follow an *Advisory* recommendation does not necessarily result in a software bug; the risk of a direct correlation between non-conformance of an advisory rule and a software bug is low. Non-compliance with an advisory recommendation level does not require a supporting technical explanation, however, as the quality of the code may be impacted, the reason for the non-conformance should be understood.

93.2 Remediation

Remediation indicates the the level of difficulty to modify/update code that does not follow this particular rule.

High

Failure to follow this rule will likely cause an unreasonable amount of modifications/updates to bring the code base into compliance.

Medium

Failure to follow this rule will likely cause a large amount of modifications/updates to bring the code base into compliance, but those changes may still be cost-effective.

Low

Failure to follow this rule may cause a small amount of modifications/updates

to bring the code base into compliance, but those changes will be minor compared to the benefit.

N/A

This rule is more of a design decision (as opposed to a coding flaw) and therefore, if the rule is violated, it is done so with a specific purpose.

DYNAMIC STORAGE MANAGEMENT (DYN)

Goal

Maintainability

✓

Reliability

✓

Portability
Performance

✓

Security

✓

Description

Have a plan for managing dynamic memory allocation and deallocation.

Rules

DYN01, DYN02, DYN03, DYN04, DYN05, DYN06

In Ada, objects are created by being either *declared* or *allocated*. Declared objects may be informally thought of as being created "on the stack" although such details are not specified by the language. *Allocated* objects may be thought of as being allocated "from the heap", which is, again, an informal term. Allocated objects are created by the evaluation of allocators represented by the reserved word **new** and, unlike declared objects, have lifetimes independent of scope.

The terms *static* and *dynamic* tend to be used in place of *declared* and *allocated*, although in traditional storage management terminology all storage allocation in Ada is dynamic. In the following discussion, the term *dynamic allocation* refers to storage that is allocated by allocators. *Static* object allocation refers to objects that are declared. *Deallocation* refers to the reclamation of allocated storage.

Unmanaged dynamic storage allocation and deallocation can lead to storage exhaustion; the required analysis is difficult under those circumstances. Furthermore, access values can establish aliases that complicate verification, and explicit deallocation of dynamic storage can lead to specific errors (e.g., "double free", "use after free") having unpredictable results. As a result, the prevalent approach to storage management in high-integrity systems is to disallow dynamic management techniques completely. [SEI-C] [MISRA2013] [Holzmann2006] [ISO2000]

However, restricted forms of storage management and associated feature usage can support the necessary reliability and analyzability characteristics while retaining sufficient expressive power to justify the analysis expense. The following sections present possible approaches, including the traditional approach in which no dynamic behavior is allowed. Individual projects may then choose which storage management approach best fits their requirements and apply appropriate tailoring, if necessary, to the specific guidelines.

Realization

There is a spectrum of management schemes possible, trading ease of analysis against

increasing expressive power. At one end there is no dynamic memory allocation (and hence, deallocation) allowed, making analysis trivial. At the other end, nearly the full expressive power of the Ada facility is available, but with analyzability partially retained. In the latter, however, the user must create the allocators in such a manner as to ensure proper behavior.

Rule DYN01 is Required, as it avoids problematic features whatever the strategy chosen. Rules DYN02-05 are marked as Advisory, because one of them should be chosen and enforced throughout a given software project.

94.1 Common High Integrity Restrictions (DYN01)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

Performance

Security

✓

Remediation → Low

Verification Method → Compiler restrictions

94.1.1 Reference

Ada Reference Manual: H.4 High Integrity Restrictions⁴⁵¹

94.1.2 Description

The following restrictions must be in effect:

- No_Anonymous_Allocators
- No_Coextensions
- No_Access_Parameter_Allocators
- Immediate_Reclamation

The first three restrictions prevent problematic usage that, for example, may cause unreclaimed (and unreclaimable) storage. The last restriction ensures any storage allocated by the compiler at run-time for representing objects is reclaimed at once. (That restriction does not apply to objects created by allocators in the application.)

⁴⁵¹ <http://www.ada-auth.org/standards/12rm/html/RM-H-4.html>

94.1.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

94.1.4 Applicable Common Weakness Enumeration

- **CWE-401 - Missing Release of Memory after Effective Lifetime**⁴⁵²
- **CWE-415 - Double Free**⁴⁵³
- **CWE-416 - Use After Free**⁴⁵⁴

94.1.5 Noncompliant Code Example

For No_Anonymous_Allocators:

```
X : access String := new String'("Hello");
...
X := new String'("Hello");
```

For No_Coextensions:

```
type Object (Msg : access String) is ...
Obj : Object (Msg => new String'("Hello"));
```

For No_Access_Parameter_Allocators:

```
procedure P (Formal : access String);
...
P (Formal => new String'("Hello"));
```

94.1.6 Compliant Code Example

For No_Anonymous_Allocators, use a named access type:

```
type String_Reference is access all String;
S : constant String_Reference := new String'("Hello");
X : access String := S;
...
X := S;
```

For No_Coextensions, use a variable of a named access type:

```
type Object (Msg : access String) is ...
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
Obj : Object (Msg => S);
```

For No_Access_Parameter_Allocators, use a variable of a named access type:

```
procedure P (Formal : access String);
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
...
P (Formal => S);
```

⁴⁵² <https://cwe.mitre.org/data/definitions/401.html>

⁴⁵³ <https://cwe.mitre.org/data/definitions/415.html>

⁴⁵⁴ <https://cwe.mitre.org/data/definitions/416.html>

94.1.7 Notes

The compiler will detect violations of the first three restrictions. Note that GNATcheck can detect violations in addition to the compiler.

The fourth restriction is a directive for implementation behavior, not subject to source-based violation detection.

94.2 Traditional Static Allocation Policy (DYN02)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

Performance

Security

✓

Remediation → Low

Verification Method → Compiler restrictions

94.2.1 Reference

MISRA C Dir 4.12: "Dynamic memory allocation shall not be used."

94.2.2 Description

The following restrictions must be in effect:

- No_Allocators
- No_Task_Allocators

Under the traditional approach, no dynamic allocations and no deallocations occur. Only declared objects are used and no access types of any kind appear in the code.

Without allocations there is no issue with deallocation as there would be nothing to deallocate. *Heap* storage exhaustion and fragmentation are clearly prevented although storage may still be exhausted due to insufficient stack size allotments.

In this approach the following constructs are not allowed:

- Allocators
- Access-to-constant access types
- Access-to-variable access types
- User-defined storage pools
- Unchecked Deallocations

94.2.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

94.2.4 Applicable Common Weakness Enumeration

- **CWE-401 - Missing Release of Memory after Effective Lifetime**⁴⁵⁵
- **CWE-758 - Reliance on Undefined, Unspecified, or Implementation-Defined Behavior**⁴⁵⁶
- **CWE-771 - Missing Reference to Active Allocated Resource**⁴⁵⁷
- **CWE-1325 - Improperly Controlled Sequential Memory Allocation**⁴⁵⁸

94.2.5 Noncompliant Code Example

Any code using the constructs listed above.

94.2.6 Compliant Code Example

N/A

94.2.7 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

94.3 Access Types Without Allocators Policy (DYN03)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

Performance

Security

✓

Remediation → Low

Verification Method → Compiler restrictions

⁴⁵⁵ <https://cwe.mitre.org/data/definitions/401.html>

⁴⁵⁶ <https://cwe.mitre.org/data/definitions/758.html>

⁴⁵⁷ <https://cwe.mitre.org/data/definitions/771.html>

⁴⁵⁸ <https://cwe.mitre.org/data/definitions/1325.html>

94.3.1 Reference

MISRA C Rule 21.3: "The memory allocation and deallocation functions of <stdlib.h> shall not be used."

94.3.2 Description

The following restrictions must be in effect:

- No_Allocators
- No_Dependence => Ada.Unchecked_Deallocation

In this approach dynamic access values are only created via the attribute 'Access applied to aliased objects. Allocation and deallocation never occur. As a result, storage exhaustion cannot occur because no *dynamic* allocations occur. Fragmentation cannot occur because there are no deallocations.

In this approach the following constructs are not allowed:

- Allocators
- User-defined storage pools
- Unchecked Deallocations

Aspects should be applied to all access types in this approach, specifying a value of zero for the storage size. Although the restriction No_Allocators is present, such clauses may be necessary to prevent any default storage pools from being allocated for the access types, even though the pools would never be used. A direct way to accomplish this is to use `pragma Default_Storage_Pool` with a parameter of `null` like so:

```
pragma Default_Storage_Pool (null);
```

The above would also ensure no allocations can occur with access types that have the default pool as their associated storage pool (per [Ada Reference Manual: 13.11.3 \(6.1/3\) Default Storage Pools](#)⁴⁵⁹).

94.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.14 Dangling reference to heap [XYK]

94.3.4 Applicable Common Weakness Enumeration

- **CWE-401 - Missing Release of Memory after Effective Lifetime**⁴⁶⁰
- **CWE-415 - Double Free**⁴⁶¹
- **CWE-416 - Use After Free**⁴⁶²
- **CWE-771 - Missing Reference to Active Allocated Resource**⁴⁶³
- **CWE-1325 - Improperly Controlled Sequential Memory Allocation**⁴⁶⁴

⁴⁵⁹ <http://www.ada-auth.org/standards/12rm/html/RM-13-11-3.html>

⁴⁶⁰ <https://cwe.mitre.org/data/definitions/401.html>

⁴⁶¹ <https://cwe.mitre.org/data/definitions/415.html>

⁴⁶² <https://cwe.mitre.org/data/definitions/416.html>

⁴⁶³ <https://cwe.mitre.org/data/definitions/771.html>

⁴⁶⁴ <https://cwe.mitre.org/data/definitions/1325.html>

94.3.5 Noncompliant Code Example

Any code using the constructs listed above.

94.3.6 Compliant Code Example

```
type Descriptor is ...;
type Descriptor_Ref is access all Descriptor;
...
Device : aliased Descriptor;
...
P : Descriptor_Ref := Device'Access;
...
```

94.3.7 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

94.4 Minimal Dynamic Allocation Policy (DYN04)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

Performance

Security

Remediation → Low

Verification Method → Compiler restrictions

94.4.1 Reference

Power of Ten rule 3: "Do not use dynamic memory allocation after initialization."

94.4.2 Description

The following restrictions must be in effect:

- No_Local_Allocators
- No_Dependence => Ada.Unchecked_Deallocation

In this approach dynamic allocation is only allowed during "start-up" and no later. Deallocation never occur. As a result, storage exhaustion should never occur assuming the initial allotment is sufficient. This assumption is as strong as when using only declared objects on

the "stack" because in that case a sufficient initial storage allotment for the stack must be made.

In this approach the following constructs are not allowed:

- Unchecked Deallocations

Note that some operating systems intended for this domain directly support this policy.

94.4.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

94.4.4 Applicable Common Weakness Enumeration

- **CWE-401 - Missing Release of Memory after Effective Lifetime**⁴⁶⁵
- **CWE-415 - Double Free**⁴⁶⁶
- **CWE-416 - Use After Free**⁴⁶⁷
- **CWE-459 - Incomplete Cleanup**⁴⁶⁸
- **CWE-771 - Missing Reference to Active Allocated Resource**⁴⁶⁹
- **CWE-1325 - Improperly Controlled Sequential Memory Allocation**⁴⁷⁰

94.4.5 Noncompliant Code Example

Any code using the constructs listed above.

94.4.6 Compliant Code Example

Code performing dynamic allocations any time prior to an arbitrary point designated as the end of the "startup" interval.

94.4.7 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

94.5 User-Defined Storage Pools Policy (DYN05)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

⁴⁶⁵ <https://cwe.mitre.org/data/definitions/401.html>

⁴⁶⁶ <https://cwe.mitre.org/data/definitions/415.html>

⁴⁶⁷ <https://cwe.mitre.org/data/definitions/416.html>

⁴⁶⁸ <https://cwe.mitre.org/data/definitions/459.html>

⁴⁶⁹ <https://cwe.mitre.org/data/definitions/771.html>

⁴⁷⁰ <https://cwe.mitre.org/data/definitions/1325.html>

Reliability



Portability
Performance
Security



Remediation → Low

Verification Method → Code inspection

94.5.1 Reference

MISRA C Rule 21.3: "The memory allocation and deallocation functions of <stdlib.h> shall not be used."

94.5.2 Description

There are two issues that make storage utilization analysis difficult:

1. the predictability of the allocation and deallocation implementation, and
2. how access values are used by the application.

The behavior of the underlying implementation is largely undefined and may, for example, consist of calls to the operating system (if present). Application code can manipulate access values beyond the scope of analysis.

Under this policy, the full expressive power of access-to-object types is provided but one of the two areas of analysis difficulty is removed. Specifically, predictability of the allocation and deallocation implementation is achieved via user-defined storage pools. With these storage pools, the implementation of allocation (**new**) and deallocation (instances of Ada.Unchecked_Deallocation) is defined by the pool type.

If the pool type is implemented with fixed-size blocks on the stack, allocation and deallocation timing behavior are predictable.

Such an implementation would also be free from fragmentation.

Given an analysis providing the worst-case allocations and deallocations, it would be possible to verify that pool exhaustion does not occur. However, as mentioned such analysis can be quite difficult. A mitigation would be the use of the "owning" access-to-object types provided by SPARK.

In this approach no storage-related constructs are disallowed unless the SPARK subset is applied.

94.5.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

94.5.4 Applicable Common Weakness Enumeration

- **CWE-401 - Missing Release of Memory after Effective Lifetime**⁴⁷¹
- **CWE-415 - Double Free**⁴⁷²
- **CWE-416 - Use After Free**⁴⁷³

⁴⁷¹ <https://cwe.mitre.org/data/definitions/401.html>

⁴⁷² <https://cwe.mitre.org/data/definitions/415.html>

⁴⁷³ <https://cwe.mitre.org/data/definitions/416.html>

- **CWE-459 - Incomplete Cleanup**⁴⁷⁴

94.5.5 Noncompliant Code Example

Allocation via an access type not tied to a user-defined storage pool.

94.5.6 Compliant Code Example

```
Heap : Sequential_Fixed_Blocks.Storage_Pool
      (Storage_Size => Required_Storage_Size,
       Element_Size => Representable_Obj_Size,
       Alignment    => Representation_Alignment);
type Pointer is access all Unsigned_Longword with
  Storage_Pool => Heap;
Ptr : Pointer;
...
Ptr := new Unsigned_Longword; -- from Heap
```

94.5.7 Notes

Enforcement of this approach can only be provided by manual code review unless SPARK is used.

However, the User-Defined Storage Pools Policy can be enforced statically by specifying `Default_Storage_Pool (null)`. This essentially requires all access types to have a specified storage pool if any allocators are used with the access type.

94.6 Statically Determine Maximum Stack Requirements (DYN06)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability
Performance
Security

Remediation → Low

Verification Method → Static analysis tools

⁴⁷⁴ <https://cwe.mitre.org/data/definitions/459.html>

94.6.1 Reference

N/A

94.6.2 Description

Each Ada application task has a stack, as does the "environment task" that elaborates library packages and calls the main subprogram. A tool to statically determine the maximum storage required for these stacks must be used, per task.

This guideline concerns another kind of dynamic memory utilization. The previous guidelines concerned the management of storage commonly referred to as the "heap." This guideline concerns the storage commonly referred to as the "stack." (Neither term is defined by the language, but both are commonly recognized and are artifacts of the underlying run-time library or operating system implementation.)

94.6.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

94.6.4 Applicable Common Weakness Enumeration

- **CWE-770 - Allocation of Resources Without Limits or Throttling**⁴⁷⁵
- **CWE-789 - Uncontrolled Memory Allocation**⁴⁷⁶

94.6.5 Noncompliant Code Example

N/A

94.6.6 Compliant Code Example

N/A

94.6.7 Notes

The **GNATstack**⁴⁷⁷ tool can statically determine the maximum requirements per task.

⁴⁷⁵ <https://cwe.mitre.org/data/definitions/770.html>

⁴⁷⁶ <https://cwe.mitre.org/data/definitions/789.html>

⁴⁷⁷ http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html

SAFE RECLAMATION (RCL)

Goal

Maintainability



Reliability



Portability
Performance



Security



Description

Related to managing dynamic storage at the system (policy) level, these statement-level rules concern the safe reclamation of access (*pointer*) values.

Rules

RCL01, RCL02, RCL03

95.1 No Multiple Reclamations (RCL01)

Level → Mandatory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



Performance
Security



Remediation → High

Verification Method → Code inspection

95.1.1 Reference

N/A

95.1.2 Description

Never deallocate the storage designated by a given access value more than once.

95.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

95.1.4 Applicable Common Weakness Enumeration

- **CWE-415 - Double Free**⁴⁷⁸
- **CWE-416 - Use After Free**⁴⁷⁹

95.1.5 Noncompliant Code Example

```
type String_Reference is access all String;
procedure Free is new Ada.Unchecked_Deallocation
  (Object => String, Name => String_Reference);
S : String_Reference := new String'("Hello");
Y : String_Reference;
begin
  Y := S;
  Free (S);
  Free (Y);
```

95.1.6 Compliant Code Example

Remove the call to Free (Y).

95.1.7 Notes

Enforcement of this rule can be provided by manual code review, unless deallocation is forbidden via No_Unchecked_Deallocation or SPARK is used, as ownership analysis in SPARK detects such cases. Note that storage utilization analysis tools such as Valgrind can usually find this sort of error. In addition, a GNAT-defined storage pool is available to help debug such errors.

95.2 Only Reclaim Allocated Storage (RCL02)

Level → Mandatory

Category

Safety

✓

Cyber

✓

Goal

⁴⁷⁸ <https://cwe.mitre.org/data/definitions/415.html>

⁴⁷⁹ <https://cwe.mitre.org/data/definitions/416.html>

Maintainability

✓

Reliability

✓

Portability

✓

**Performance
Security**

✓

Remediation → High**Verification Method** → Code inspection

95.2.1 Reference

[SEI-C] MEM34-C: Only Free Memory Allocated Dynamically

95.2.2 Description

Only deallocate storage that was dynamically allocated by the evaluation of an allocator (i.e., **new**).

This is possible because Ada allows creation of access values designating declared (aliased) objects.

95.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

95.2.4 Applicable Common Weakness Enumeration

- **CWE-590 - Free of Memory not on the Heap**⁴⁸⁰

95.2.5 Noncompliant Code Example

```

type String_Reference is access all String;
procedure Free is new Ada.Unchecked_Deallocation
  (Object => String, Name => String_Reference);
S : aliased String := "Hello";
Y : String_Reference := S'Access;
begin
  Free (Y);

```

95.2.6 Compliant Code Example

Remove the call to Free (Y).

95.2.7 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via No_Unchecked_Deallocation.

⁴⁸⁰ <https://cwe.mitre.org/data/definitions/590.html>

95.3 Only Reclaim to the Same Pool (RCL03)

Level → Mandatory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



Performance

Security



Remediation → High

Verification Method → Code inspection

95.3.1 Reference

N/A

95.3.2 Description

When deallocating, ensure that the pool to which the storage will be returned is the same pool from which it was allocated. Execution is erroneous otherwise, meaning anything can happen (Ada Reference Manual: 13.11.2 (16) [Unchecked Storage Deallocation](#)⁴⁸¹).

Each access type has an associated storage pool, either implicitly by default, or explicitly with a storage pool specified by the programmer. The implicit default pool might not be the same pool used for another access type, even an access type designating the same subtype.

95.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

95.3.4 Applicable Common Weakness Enumeration

- [CWE-401 - Missing Release of Memory after Effective Lifetime](#)⁴⁸²

95.3.5 Noncompliant Code Example

⁴⁸¹ <http://www.ada-auth.org/standards/12rm/html/RM-13-11-2.html>

⁴⁸² <https://cwe.mitre.org/data/definitions/401.html>

```

type Pointer1 is access all Integer;
type Pointer2 is access all Integer;
P1 : Pointer1;
P2 : Pointer2;
procedure Free is new Ada.Unchecked_Deallocation
  (Object => Integer, Name => Pointer2);
begin
  P1 := new Integer;
  P2 := Pointer2 (P1);
  Call_Something ( P2.all );
  ...
  Free (P2);

```

In the above, P1.all was allocated from Pointer1's [Storage_Pool](#), but, via the type conversion, the code above is attempting to return it to Pointer2's [Storage_Pool](#), which may be a different pool.

95.3.6 Compliant Code Example

```

type Pointer1 is access all Integer;
type Pointer2 is access all Integer;
P1 : Pointer1;
P2 : Pointer2;
procedure Free is new Ada.Unchecked_Deallocation
  (Object => Integer, Name => Pointer1);
begin
  P1 := new Integer;
  P2 := Pointer2 (P1);
  Call_Something ( P2.all );
  ...
  Free (P1);

```

95.3.7 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via `No_Unchecked_Deallocation`.

CONCURRENCY (CON)

Goal

Maintainability



Reliability



Portability



Performance



Security

Description

Have a plan for managing the use of concurrency in high-integrity applications having real-time requirements.

Rules

CON01, CON02, CON03

The canonical approach to applications having multiple periodic and aperiodic activities is to map those activities onto independent tasks, i.e., threads of control. The advantages for the application are both a matter of software engineering and also ease of implementation. For example, when the different periods are not harmonics of one another, the fact that each task executes independently means that the differences are trivially represented. In contrast, such periods are not easily implemented in a cyclic scheduler, which, by definition, involves only one (implicit) thread of control with one frame rate.

High integrity applications are subject to a number of stringent analyses, including, for example, safety analyses and certification against rigorous industry standards. In addition, high integrity applications with real-time requirements must undergo timing analysis because they must be shown to meet deadlines prior to deployment — failure to meet hard deadlines is unacceptable in this domain.

These analyses are applied both to the application and to the implementation of the underlying run-time library. However, analysis of the complete set of general Ada tasking features is not tractable, neither technically nor in terms of cost. A subset of the language is required.

The Ravenscar profile [[AdaRM2016](#)] is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are specifically intended:

- Hard real-time applications requiring predictability;
- Safety-critical systems requiring formal, stringent certification;
- High-integrity applications requiring formal static analysis and verification;

- Embedded applications requiring both a small memory footprint and low execution overhead.

Those tasking constructs that preclude analysis at the source level or analysis of the tasking portion of the underlying run-time library are disallowed.

The Ravenscar profile is necessarily strict in terms of what it removes so that it can support the stringent analyses, such as safety analysis, that go beyond the timing analysis required for real-time applications. In addition, the strict subset facilitates that timing analysis in the first place.

However, not all high-integrity applications are amenable to expression in the Ravenscar profile subset. The Jorvik profile [AdaRM2020] is an alternative subset of the Ada concurrency facilities. It is based directly on the Ravenscar profile but removes selected restrictions in order to increase expressive power, while retaining analyzability and performance. As a result, typical idioms for protected objects can be used, for example, and relative delays statements are allowed. Timing analysis is still possible but slightly more complicated, and the underlying run-time library is slightly larger and more complex.

When the most stringent analyses are required and the tightest timing is involved, use the Ravenscar profile. When a slight increase in complexity is tolerable, i.e., in those cases not undergoing all of these stringent analyses, consider using the Jorvik profile.

96.1 Use the Ravenscar Profile (CON01)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

✓

Security

Remediation → High

Verification Method → GNATcheck rule: `uses_profile:ravenscar` (supplied with document)

Mutually Exclusive → CON02

96.1.1 Reference

Ada Reference Manual: D.13 The Ravenscar and Jorvik Profiles⁴⁸³

⁴⁸³ <http://www.ada-auth.org/standards/12rm/html/RM-D-13.html>

96.1.2 Description

The following profile must be in effect:

```
pragma Profile (Ravenscar);
```

The profile is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
    No_Abort_Statements,
    No_Dynamic_Attachment,
    No_Dynamic_CPU_Assignment,
    No_Dynamic_Priorities,
    No_Implicit_Heap_Allocations,
    No_Local_Protected_Objects,
    No_Local_Timing_Events,
    No_Protected_Type_Allocators,
    No_Relative_Delay,
    No_Requeue_Statements,
    No_Select_Statements,
    No_Specific_Termination_Handlers,
    No_Task_Allocators,
    No_Task_Hierarchy,
    No_Task_Termination,
    Simple_Barriers,
    Max_Entry_Queue_Length => 1,
    Max_Protected_Entries => 1,
    Max_Task_Entries => 0,
    No_Dependence => Ada.Asynchronous_Task_Control,
    No_Dependence => Ada.Calendar,
    No_Dependence => Ada.Execution_Time.Group_Budgets,
    No_Dependence => Ada.Execution_Time.Timers,
    No_Dependence => Ada.Synchronous_Barriers,
    No_Dependence => Ada.Task_Attributes,
    No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

96.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

96.1.4 Applicable Common Weakness Enumeration

- **CWE-362 - Concurrent Execution using Shared Resource with Improper Synchronization**⁴⁸⁴
- **CWE-366 - Race Condition within a Thread**⁴⁸⁵
- **CWE-367 - Time-of-check Time-of-use (TOCTOU) Race Condition**⁴⁸⁶

⁴⁸⁴ <https://cwe.mitre.org/data/definitions/362.html>

⁴⁸⁵ <https://cwe.mitre.org/data/definitions/366.html>

⁴⁸⁶ <https://cwe.mitre.org/data/definitions/367.html>

96.1.5 Noncompliant Code Example

Any code disallowed by the profile. Remediation is **high** because use of the facilities outside the subset can be difficult to retrofit into compliance.

```
task body Task_T is
begin
  loop
    -- Error: No_Relative_Delay
    delay 1.0;
    Put_Line ("Hello World");
  end loop;
end Task_T;
```

96.1.6 Compliant Code Example

```
task body Task_T is
  Period      : constant Time_Span := Milliseconds (10);
  Activation   : Time              := Clock;
begin
  loop
    delay until Activation;
    Put_Line ("Hello World");
    Activation := Activation + Period;
  end loop;
end Task_T;
```

96.1.7 Notes

The Ada builder will detect violations if the programmer specifies this profile or corresponding pragmas. GNATcheck also can detect violations of profile restrictions.

96.2 Use the Jorvik Profile (CON02)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

✓

Security

Remediation → High

Verification Method → GNATcheck rule: uses_profile:jorvik (supplied with document)

Mutually Exclusive → CON01

96.2.1 Reference

Ada Reference Manual: D.13 The Ravenscar and Jorvik Profiles⁴⁸⁷

96.2.2 Description

The following profile must be in effect:

```
pragma Profile (Jorvik);
```

The profile is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
    No_Abort_Statements,
    No_Dynamic_Attachment,
    No_Dynamic_CPU_Assignment,
    No_Dynamic_Priorities,
    No_Local_Protected_Objects,
    No_Local_Timing_Events,
    No_Protected_Type_Allocators,
    No_Requeue_Statements,
    No_Select_Statements,
    No_Specific_Termination_Handlers,
    No_Task_Allocators,
    No_Task_Hierarchy,
    No_Task_Termination,
    Pure_Barriers,
    Max_Task_Entries => 0,
    No_Dependence => Ada.Asynchronous_Task_Control,
    No_Dependence => Ada.Execution_Time.Group_Budgets,
    No_Dependence => Ada.Execution_Time.Timers,
    No_Dependence => Ada.Task_Attributes,
    No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

The following restrictions are part of the Ravenscar profile but **not** part of the Jorvik profile.

```
No_Implicit_Heap_Allocations
No_Relative_Delay
Max_Entry_Queue_Length => 1
Max_Protected_Entries => 1
No_Dependence => Ada.Calendar
No_Dependence => Ada.Synchronous_Barriers
```

Jorvik also replaces restriction Simple_Barriers with Pure_Barriers (a weaker requirement than the restriction Simple_Barriers).

⁴⁸⁷ <http://www.ada-auth.org/standards/12rm/html/RM-D-13.html>

96.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

96.2.4 Applicable Common Weakness Enumeration

- **CWE-362 - Concurrent Execution using Shared Resource with Improper Synchronization**⁴⁸⁸
- **CWE-366 - Race Condition within a Thread**⁴⁸⁹
- **CWE-367 - Time-of-check Time-of-use (TOCTOU) Race Condition**⁴⁹⁰

96.2.5 Noncompliant Code Example

Any code disallowed by the profile. Remediation is **high** because use of the facilities outside the subset can be difficult to retrofit into compliance.

```
task body Task_T is
begin
  -- Error: Max_Task_Entries => 0
  accept Entry_Point do
    Put_Line ("Hello World");
  end Entry_Point;
  loop
    delay 1.0;
    Put_Line ("Ping");
  end loop;
end Task_T;
```

96.2.6 Compliant Code Example

```
task body Task_T is
begin
  delay 1.0;
  Put_Line ("Hello World");
  loop
    delay 1.0;
    Put_Line ("Ping");
  end loop;
end Task_T;
```

96.2.7 Notes

The Ada builder will detect violations. GNATcheck can also detect violations.

⁴⁸⁸ <https://cwe.mitre.org/data/definitions/362.html>

⁴⁸⁹ <https://cwe.mitre.org/data/definitions/366.html>

⁴⁹⁰ <https://cwe.mitre.org/data/definitions/367.html>

96.3 Avoid Shared Variables for Inter-task Communication (CON03)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



Performance



Security

Remediation → High

Verification Method → GNATcheck rule: `Volatile_Objects_Without_Address_Clauses` (builtin rule)

96.3.1 Reference

Ada Reference Manual: D.13 The Ravenscar Profile⁴⁹¹

96.3.2 Description

Although the Ravenscar and Jorvik profiles allow the use of shared variables for inter-task communication, such use is less robust and less reliable than encapsulating shared variables within protected objects.

96.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.56 Undefined behaviour [EWF]

96.3.4 Applicable Common Weakness Enumeration

- **CWE-567 - Unsynchronized Access to Shared Data in a Multithreaded Context**⁴⁹²
- **CWE-667 - Improper Locking**⁴⁹³

⁴⁹¹ <http://www.ada-auth.org/standards/12rm/html/RM-D-13.html>

⁴⁹² <https://cwe.mitre.org/data/definitions/567.html>

⁴⁹³ <https://cwe.mitre.org/data/definitions/667.html>

96.3.5 Noncompliant Code Example

```
Global_Object : Integer
  with Volatile;
function Get return Integer is (Global_Object);
```

Note that variables marked as Atomic are also Volatile, per the [Ada Reference Manual: C.6 \(8/3\) Shared Variable Control](#)⁴⁹⁴

96.3.6 Compliant Code Example

When assigned to a memory address, a Volatile variable can be used to interact with a memory-mapped device, among other similar usages.

```
Global_Object : Integer
  with Volatile,
    Address => To_Address (16#1234_5678#);
function Get return Integer is (Global_Object);
```

96.3.7 Notes

In addition to GNATcheck, SPARK and CodePeer can also detect conflicting access to unprotected variables.

⁴⁹⁴ <http://www.ada-auth.org/standards/12rm/html/RM-C-6.html>

ROBUST PROGRAMMING PRACTICE (RPP)

Goal

Maintainability



Reliability



Portability



Performance



Security



Description

These rules promote the production of robust software.

Rules

RPP01, RPP02, RPP03, RPP04, RPP05, RPP06, RPP07, RPP07, RPP08, RPP09, RPP10,
RPP11, RPP12, RPP13, RPP14

97.1 No Use of "others" in Case Constructs (RPP01)

Level → Required

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



Performance

Security

Remediation → Low

Verification Method → GNATcheck rule: OTHERS_In_CASE_Statements (builtin rule)

97.1.1 Reference

[SEI-C] MSC01-C

97.1.2 Description

Case statement alternatives and case-expressions must not include use of the **others** discrete choice option. This rule prevents accidental coverage of a choice added after the initial case statement is written, when an explicit handler was intended for the addition.

Note that this is opposite to typical C guidelines such as [SEI-C] MSC01-C. The reason is that in C, the **default** alternative plays the role of defensive code to mitigate the switch statement's non-exhaustivity. In Ada, the **case** construct is exhaustive: the compiler statically verifies that for every possible value of the **case** expression there is a branch alternative, and there is also a dynamic check against invalid values which serves as implicit defensive code. As a result, Ada's **others** alternative doesn't play C's defensive code role and therefore a stronger guideline can be adopted.

97.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.27 Switch statements and static analysis [CLL]

97.1.4 Applicable Common Weakness Enumeration

- **CWE-478 - Missing Default Case in Multiple Condition Expression**⁴⁹⁵

97.1.5 Noncompliant Code Example

```
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'succ (C);
  when others =>
    C := Character'pred (C);
end case;
```

97.1.6 Compliant Code Example

```
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'succ (C);
  when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
    C := Character'pred (C);
end case;
```

97.1.7 Notes

N/A

⁴⁹⁵ <https://cwe.mitre.org/data/definitions/478.html>

97.2 No Enumeration Ranges in Case Constructs (RPP02)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

**Performance
Security**

Remediation → Low

Verification Method → GNATcheck rule: Enumeration_Ranges_In_CASE_Statements (builtin rule)

97.2.1 Reference

Similar to RPP01

97.2.2 Description

A range of enumeration literals must not be used as a choice in a **case** statement or a **case** expression. This includes explicit ranges (A .. B), subtypes, and the 'Range' attribute. Much like the use of **others** in **case** statement alternatives, the use of ranges makes it possible for a new enumeration value to be added but not handled with a specific alternative, when a specific alternative was intended.

97.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]

97.2.4 Applicable Common Weakness Enumeration

N/A

97.2.5 Noncompliant Code Example

```
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'Succ (C);
  when '1' .. '8' =>
    C := Character'Pred (C);
end case;
```


97.2.6 Compliant Code Example

```
case Digit_T (C) is
  when '0' | '9' =>
    C := Character'Succ (C);
  when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
    C := Character'Pred (C);
end case;
```

97.2.7 Notes

N/A

97.3 Limited Use of "others" in Aggregates (RPP03)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: OTHERS_In_Aggregates (builtin rule)

97.3.1 Reference

Similar to RPP01

97.3.2 Description

Do not use an **others** choice in an extension aggregate. In **record** and **array** aggregates, do not use an **others** choice unless it is used either to refer to all components, or to all but one component.

This guideline prevents accidental provision of a general value for a **record** component or **array** component, when a specific value was intended. This possibility includes the case in which new components are added to an existing composite type.

97.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]
- 6.27 Switch statements and static analysis [CLL]

97.3.4 Applicable Common Weakness Enumeration

- **CWE-478 - Missing Default Case in Multiple Condition Expression**⁴⁹⁶

97.3.5 Noncompliant Code Example

```

type Record_T is record
  Field1 : Integer := 1;
  Field2 : Boolean := False;
  Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                  Field3 => '2',
                  others => <>);
Arr : Array_T := ('0' .. '9' => True,
                 others => False);

```

97.3.6 Compliant Code Example

```

type Record_T is record
  Field1 : Integer := 1;
  Field2 : Boolean := False;
  Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                  others => <>);
Arr : Array_T := (others => False);

```

97.3.7 Notes

N/A

97.4 No Unassigned Mode-Out Procedure Parameters (RPP04)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

⁴⁹⁶ <https://cwe.mitre.org/data/definitions/478.html>

Reliability

✓

Portability

✓

Performance
Security

Remediation → High

Verification Method → GNATcheck rule: Unassigned_OUT_Parameters (builtin rule)

97.4.1 Reference

MISRA C Rule 9.1: "The value of an object with automatic storage duration shall not be read before it has been set."

97.4.2 Description

For any procedure, all formal parameters of mode **out** must be assigned a value if the procedure exits normally. This rule ensures that, upon a normal return, the corresponding actual parameter has a defined value. Ensuring a defined value is especially important for scalar parameters because they are passed by value, such that some value is copied out to the actual. These undefined values can be especially difficult to locate because evaluation of the actual parameter's value might not occur immediately after the call returns.

97.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CS]

97.4.4 Applicable Common Weakness Enumeration

- **CWE-457 - Use of Uninitialized Variable**⁴⁹⁷

97.4.5 Noncompliant Code Example

```
for Value_T use
  (Alpha   => 2#1#,
   Baker   => 2#10#,
   Charlie => 2#100#,
   Dog     => 2#1000#,
   Invalid => 2#1111#);

procedure Noncompliant (Register : Character;
                        Registera : out Value_T;
                        Registerb : out Value_T) is
begin
  if Register = 'A' then
    Registera := Alpha;
  end if;
end Noncompliant;
```

In the above example, some value is copied back for an output parameter as specified by Register. The other parameter is not assigned, and on return the value copied to the actual parameter may not be a valid representation for a value of the type. (We give the enumeration values a non-standard representation for the sake of illustration, i.e., to make it more likely that the undefined value is not valid.)

⁴⁹⁷ <https://cwe.mitre.org/data/definitions/457.html>

97.4.6 Compliant Code Example

```

procedure Compliant (Register : Character;
                     Registera : out Value_T;
                     Registerb : out Value_T) is
begin
    Registera := Invalid;
    Registerb := Invalid;
    if Register = 'A' then
        Registera := Alpha;
    end if;
end Compliant;

```

97.4.7 Notes

The GNATcheck rule specified above only detects a trivial case of an unassigned variable and doesn't provide a guarantee that there is no uninitialized access. It is not a replacement for a rigorous check for uninitialized access provided by advanced static analysis tools such as SPARK and CodePeer.

Note that the GNATcheck rule does not check function parameters (as of Ada 2012 functions can have **out** parameters). As a result, the better choice is either SPARK or CodePeer.

97.5 No Use of "others" in Exception Handlers (RPP05)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: OTHERS_In_Exception_Handlers (builtin rule)

97.5.1 Reference

N/A

97.5.2 Description

Much like the situation with **others** in **case** statements and **case** expressions, the use of **others** in exception handlers makes it possible to omit an intended specific handler for an exception, especially a new exception added to an existing set of handlers. As a result, a subprogram could return normally without having applied any recovery for the specific exception occurrence, which is likely a coding error.

97.5.3 Applicable Vulnerability within ISO TR 24772-2

N/A

97.5.4 Applicable Common Weakness Enumeration

- **CWE-396 - Declaration of Catch for Generic Exception**⁴⁹⁸

97.5.5 Noncompliant Code Example

```
procedure Noncompliant (X : in out Integer) is
begin
  X := X * X;
exception
  when others =>
    X := -1;
end Noncompliant;
```

97.5.6 Compliant Code Example

```
procedure Compliant (X : in out Integer) is
begin
  X := X * X;
exception
  when Constraint_Error =>
    X := -1;
end Compliant;
```

97.5.7 Notes

ISO TR 24772-2: 6.50.2 slightly contradicts this when applying exception handlers around calls to library routines:

- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler **when others =>**
- Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler **when others =>**

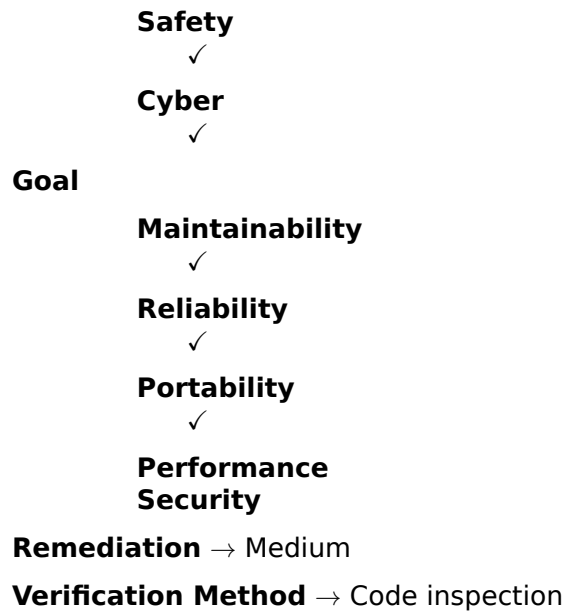
ISO TR 24772-2 also recommends "All tasks should contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions."

97.6 Avoid Function Side-Effects (RPP06)

Level → Advisory

Category

⁴⁹⁸ <https://cwe.mitre.org/data/definitions/396.html>



97.6.1 Reference

MISRA C Rule 13.2: "The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders."

97.6.2 Description

Functions cannot update an actual parameter or global variable.

A side effect occurs when evaluation of an expression updates an object. This rule applies to function calls, a specific form of expression.

Side effects enable one form of parameter aliasing (see below) and evaluation order dependencies. In general they are a potential point of confusion because the reader expects only a computation of a value.

There are useful idioms based on functions with side effects. Indeed, a random number generator expressed as a function must use side effects to update the seed value. So-called "memo" functions are another example, in which the function tracks the number of times it is called. Therefore, exceptions to this rule are anticipated but should only be allowed on a per-instance basis after careful analysis.

97.6.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

97.6.4 Applicable Common Weakness Enumeration

N/A

97.6.5 Noncompliant Code Example

```

Call_Count : Integer := 0;
function F return Boolean is
  Result : Boolean;
begin
  ...

```

(continues on next page)

(continued from previous page)

```
    Call_Count := Call_Count + 1;  
    return Result;  
end F;
```

97.6.6 Compliant Code Example

Remove the update to `Call_Count`, or change the function into a procedure with a parameter for `Call_Count`.

97.6.7 Notes

Violations are detected by SPARK as part of a rule disallowing side effects on expression evaluation.

97.7 Functions Only Have Mode "in" (RPP07)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `function_out_parameters` (supplied with document)

97.7.1 Reference

N/A

97.7.2 Description

Functions must have only mode `in`.

As of Ada 2012, functions are allowed to have the same modes as procedures. However, this can lead to side effects and aliasing.

This rule disallows all modes except mode `in` for functions.

97.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

97.7.4 Applicable Common Weakness Enumeration

N/A

97.7.5 Noncompliant Code Example

```
function Noncompliant (Value : in out Integer) return Integer is
begin
  if Value < Integer'last then
    Value := Value + 1;
  end if;
  return Value;
end Noncompliant;
```

97.7.6 Compliant Code Example

```
function Compliant (Value : Integer) return Integer is
begin
  return Value + 1;
end Compliant;
```

OR

```
procedure Compliant (Value : in out Integer) is
begin
  if Value < Integer'last then
    Value := Value + 1;
  end if;
end Compliant;
```

97.7.7 Notes

Violations are detected by SPARK.

97.8 Limit Parameter Aliasing (RPP08)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance Security

Remediation → High

Verification Method → Code inspection

97.8.1 Reference

Ada Reference Manual: 6.2 Formal Parameter Modes⁴⁹⁹

SPARK Reference Manual: Anti-Aliasing⁵⁰⁰

97.8.2 Description

In software, an alias is a name which refers to the same object as another name. In some cases, it is an error in Ada to reference an object through a name while updating it through another name in the same subprogram. Most of these cases cannot be detected by a compiler. Even when not an error, the presence of aliasing makes it more difficult to understand the code for both humans and analysis tools, and thus it may lead to errors being introduced during maintenance.

This rule is meant to detect problematic cases of aliasing that are introduced through the actual parameters and between actual parameters and global variables in a subprogram call. It is a simplified version of the SPARK rule for anti-aliasing defined in [SPARK Reference Manual, section 6.4.2: Anti-Aliasing](#)⁵⁰¹.

A formal parameter is said to be immutable when the subprogram cannot modify its value or modify the value of an object by dereferencing a part of the parameter of access type (at any depth in the case of SPARK). In Ada and SPARK, this corresponds to either an anonymous access-to-constant parameter or a parameter of mode **in** and not of an access type. Otherwise, the formal parameter is said to be mutable.

A procedure call shall not pass two actual parameters which potentially introduce aliasing via parameter passing unless either:

- both of the corresponding formal parameters are immutable; or
- at least one of the corresponding formal parameters is immutable and is of a by-copy type that is not an access type.

If an actual parameter in a procedure call and a global variable referenced by the called procedure potentially introduce aliasing via parameter passing, then:

- the corresponding formal parameter shall be immutable; and
- if the global variable is written in the subprogram, then the corresponding formal parameter shall be of a by-copy type that is not an access type.

Where one of the rules above prohibits the occurrence of an object or any of its subcomponents as an actual parameter, the following constructs are also prohibited in this context:

- A type conversion whose operand is a prohibited construct;
- A call to an instance of `Unchecked_Conversion` whose operand is a prohibited construct;
- A qualified expression whose operand is a prohibited construct;
- A prohibited construct enclosed in parentheses.

⁴⁹⁹ <http://www.ada-auth.org/standards/12rm/html/RM-6-2.html>

⁵⁰⁰ <https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing>

⁵⁰¹ <https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing>

97.8.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CSJ]

97.8.4 Applicable Common Weakness Enumeration

N/A

97.8.5 Noncompliant Code Example

```

type R is record
  Data : Integer := 0;
end record;

procedure Detect_Aliasing (Val_1 : in out R;
                          Val_2 : in R)
is
begin
  null;
end Detect_Aliasing;

Obj : R;

begin
  Detect_Aliasing (Obj, Obj);

```

97.8.6 Compliant Code Example

Do not pass Obj as the actual parameter to both formal parameters.

97.8.7 Notes

All violations are detected by SPARK. The GNAT compiler switch -gnateA[1] enables detection of some cases, but not all.

97.9 Use Precondition and Postcondition Contracts (RPP09)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance Security



Remediation → Low

Verification Method → Code inspection

97.9.1 Reference

Power of Ten rule 5: "The assertion density of the code should average to a minimum of two assertions per function."

97.9.2 Description

Subprograms should declare Pre and/or Post contracts. Developers should consider specifying the Global contract as well, when the default does not apply.

Subprogram contracts complete the Ada notion of a specification, enabling clients to know what the subprogram does without having to know how it is implemented.

Preconditions define those logical (Boolean) conditions required for the body to be able to provide the specified behavior. As such, they are obligations on the callers. These conditions are checked at run-time in Ada, prior to each call, and verified statically in SPARK.

Postconditions define those logical (Boolean) conditions that will hold after the call returns normally. As such, they express obligations on the implementer, i.e., the subprogram body. The implementation must be such that the postcondition holds, either at run-time for Ada, or statically in SPARK.

Not all subprograms will have both a precondition and a postcondition, some will have neither.

The Global contract specifies interactions with those objects not local to the corresponding subprogram body. As such, they help complete the specification because, otherwise, one would need to examine the body of the subprogram itself and all those it calls, directly or indirectly, to know whether any global objects were accessed.

97.9.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

97.9.4 Applicable Common Weakness Enumeration

- **CWE-754 - Improper Check for Unusual or Exceptional Conditions**⁵⁰²

97.9.5 Noncompliant Code Example

```
type Stack is private;  
procedure Push (This : in out Stack; Item : Element);
```

97.9.6 Compliant Code Example

```
type Stack is private;  
procedure Push (This : in out Stack; Item : Element) with  
  Pre => not Full (This),
```

(continues on next page)

⁵⁰² <https://cwe.mitre.org/data/definitions/754.html>

(continued from previous page)

```

Post => not Empty (This)
      and Top_Element (This) = Item
      and Extent (This) = Extent (This)'Old + 1
      and Unchanged (This'Old, Within => This),
Global => null;

```

97.9.7 Notes

This rule must be enforced by manual inspection.

Moreover, the program must be compiled with enabled assertions (GNAT -gnata switch) to ensure that the contracts are executed, or a sound static analysis tool such as CodePeer or SPARK toolset should be used to prove that the contracts are always true.

97.10 Do Not Re-Verify Preconditions in Subprogram Bodies (RPP10)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

**Performance
Security**

Remediation → Low

Verification Method → Static analysis tools

97.10.1 Reference

N/A

97.10.2 Description

Do not re-verify preconditions in the corresponding subprogram bodies. It is a waste of cycles and confuses the reader as well.

97.10.3 Applicable Vulnerability within ISO TR 24772-2

N/A

97.10.4 Applicable Common Weakness Enumeration

N/A

97.10.5 Noncompliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack; Item : Element) with
  Pre  => not Full (This),
  Post => ...
...
procedure Push (This : in out Stack; Item : Element) is
begin
  if Full (This) then -- redundant check
    raise Overflow;
  end if;
  This.Top := This.Top + 1;
  This.Values (This.Top) := Item;
end Push;
```

97.10.6 Compliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack; Item : Element) with
  Pre  => not Full (This),
  Post => ...
...
procedure Push (This : in out Stack; Item : Element) is
begin
  This.Top := This.Top + 1;
  This.Values (This.Top) := Item;
end Push;
```

97.10.7 Notes

This rule can be enforced by CodePeer or SPARK, via detection of dead code.

97.11 Always Use the Result of Function Calls (RPP11)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance Security

Remediation → Low

Verification Method → Compiler restrictions

97.11.1 Reference

MISRA C Rule 17.7: "The value returned by a function having non-void return type shall be used," and

Directive 4.7: "If a function returns error information, that error information shall be tested."

97.11.2 Description

In Ada and SPARK, it is not possible to ignore the object returned by a function call. The call must be treated as a value, otherwise the compiler will reject the call. For example, the value must be assigned to a variable, or passed as the actual parameter to a formal parameter of another call, and so on.

However, that does not mean that the value is actually used to compute some further results. Although almost certainly a programming error, one could call a function, assign the result to a variable (or constant), and then not use that variable further.

Note that functions will not have side-effects (due to RPP06) so it is only the returned value that is of interest here.

97.11.3 Applicable Vulnerability within ISO TR 24772-2

- 6.47 Inter-language calling [DJS]

97.11.4 Applicable Common Weakness Enumeration

- **CWE-252 - Unchecked Return Value**⁵⁰³
- **CWE-563 - Assignment to Variable without Use**⁵⁰⁴

97.11.5 Noncompliant Code Example

N/A

97.11.6 Compliant Code Example

N/A

97.11.7 Notes

The GNAT compiler warning switch `-gnatwu` (or the more general `-gnatwa` warnings switch) will cause the compiler to detect variables assigned but not read. CodePeer will detect these unused variables as well. SPARK goes further by checking that all computations contribute all the way to subprogram outputs.

⁵⁰³ <https://cwe.mitre.org/data/definitions/252.html>

⁵⁰⁴ <https://cwe.mitre.org/data/definitions/563.html>

97.12 No Recursion (RPP12)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



**Performance
Security**

Remediation → Low

Verification Method → GNATcheck rule: Recursive_Subprograms (builtin rule)

97.12.1 Reference

MISRA C Rule 17.2: "Functions shall not call themselves, either directly or indirectly."

97.12.2 Description

No subprogram shall be invoked, directly or indirectly, as part of its own execution.

In addition to making static analysis more complex, recursive calls make static stack usage analysis extremely difficult, requiring, for example, manual supply of call limits.

97.12.3 Applicable Vulnerability within ISO TR 24772-2

- 6.35 Recursion [GDL]

97.12.4 Applicable Common Weakness Enumeration

- **CWE-674 - Uncontrolled Recursion**⁵⁰⁵

97.12.5 Noncompliant Code Example

```
function Noncompliant (N : Positive) return Positive is
begin
  if N = 1 then
    return 1;
  else
    return N * Noncompliant (N - 1); -- could overflow
  end if;
end Noncompliant;
```

⁵⁰⁵ <https://cwe.mitre.org/data/definitions/674.html>

97.12.6 Compliant Code Example

```
function Compliant (N : Positive) return Positive is
  Result : Positive := 1;
begin
  for K in 2 .. N loop
    Result := Result * K;  -- could overflow
  end loop;
  return Result;
end Compliant;
```

97.12.7 Notes

The compiler will detect violations with the restriction `No_Recursion` in place. Note this is a dynamic check.

The GNATcheck rule specified above is a static check, subject to the limitations described in [GNATcheck Reference Manual: Recursive Subprograms](#)⁵⁰⁶.

97.13 No Reuse of Standard Typemarks (RPP13)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

**Performance
Security**

Remediation → Low

Verification Method → GNATcheck rule: `overrides_standard_name` (supplied with document)

97.13.1 Reference

N/A

97.13.2 Description

Do not reuse the names of standard Ada typemarks (e.g. `type Integer is range -1_000 .. 1_000;`)

⁵⁰⁶ https://docs.adacore.com/live/wave/lkql/html/gnatcheck_rm/gnatcheck_rm/predefined_rules.html#recursive-subprograms

When a developer uses an identifier that has the same name as a standard typemark, such as **Integer**, a subsequent maintainer might be unaware that this identifier does not actually refer to Standard. **Integer** and might unintentionally use the locally-scoped **Integer** rather than the original Standard. **Integer**. The locally-scoped **Integer** can have different attributes (and may not even be of the same base type).

97.13.3 Applicable Vulnerability within ISO TR 24772-2

N/A

97.13.4 Applicable Common Weakness Enumeration

- **CWE-843 - Access of Resource Using Incompatible Type ('Type Confusion')**⁵⁰⁷

97.13.5 Noncompliant Code Example

```
type Boolean is range 0 .. 1 with Size => 1;  
type Character is ('A', 'E', 'I', 'O', 'U');
```

97.13.6 Compliant Code Example

```
type Boolean_T is range 0 .. 1 with Size => 1;  
type Character_T is ('A', 'E', 'I', 'O', 'U');
```

97.13.7 Notes

N/A

97.14 Use Symbolic Constants for Literal Values (RPP14)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: Numeric_Literals (builtin rule)

⁵⁰⁷ <https://cwe.mitre.org/data/definitions/843.html>

97.14.1 Reference

N/A

97.14.2 Description

Extensive use of literals in a program can lead to two problems. First, the meaning of the literal is often obscured or unclear from the context. Second, changing a frequently used literal requires searching the entire program source for that literal and distinguishing the uses that must be modified from those that should remain unmodified.

Avoid these problems by declaring objects with meaningfully named constants, setting their values to the desired literals, and referencing the constants instead of the literals throughout the program. This approach clearly indicates the meaning or intended use of each literal. Furthermore, should the constant require modification, the change is limited to the declaration; searching the code is unnecessary.

Some literals can be replaced with attribute values. For example, when iterating over an array, it is better to use `Array_Object'First .. Array_Object'Last` than using `1 .. Array_Object'Length`.

97.14.3 Applicable Vulnerability within ISO TR 24772-2

N/A

97.14.4 Applicable Common Weakness Enumeration

- **CWE-547 - Use of Hard-coded, Security-relevant Constants**⁵⁰⁸
- **CWE-1106 - Insufficient Use of Symbolic Constants**⁵⁰⁹

97.14.5 Noncompliant Code Example

```
type Array_T is array (0 .. 31) of Boolean;
function Any_Set (X : Array_T) return Boolean is
  (for some Flag in 0 .. 31 => X (Flag));
```

97.14.6 Compliant Code Example

```
Number_Of_Bits : constant := 32;
type Array_T is array (0 .. Number_Of_Bits - 1) of Boolean;
function Any_Set (X : Array_T) return Boolean is
  (for some Flag in X'Range => X (Flag));
```

97.14.7 Notes

N/A

⁵⁰⁸ <https://cwe.mitre.org/data/definitions/547.html>

⁵⁰⁹ <https://cwe.mitre.org/data/definitions/1106.html>

EXCEPTION USAGE (EXU)

Goal

Maintainability



Reliability



Portability



Performance



Security



Description

Have a plan for managing the use of Ada exceptions at the application level.

Rules

EXU01, EXU02, EXU03, EXU04

Exceptions in modern languages present the software architect with a dilemma. On one hand, exceptions can increase integrity by allowing components to signal specific errors in a manner that cannot be ignored, and, in general, allow residual errors to be caught. (Although there should be no unexpected errors in high integrity code, there may be some such errors due, for example, to unforeseeable events such as radiation-induced single-event upsets.) On the other hand, unmanaged use of exceptions increases verification expense and difficulty, especially flow analysis, perhaps to an untenable degree. In that case overall integrity is reduced or unwarranted.

In addition, programming languages may define some system-level errors in terms of language-defined exceptions. Such exceptions may be unavoidable, at least at the system level. For example, in Ada, stack overflow is signalled with the language-defined `Storage_Error` exception. Other system events, such as bus error, may also be mapped to language-defined or vendor-defined exceptions.

Complicating the issue further is the fact that, if exceptions are completely disallowed, there will be no exception handling code in the underlying run-time library. The effects are unpredictable if any exception actually does occur.

Therefore, for the application software the system software architect must decide whether to allow exceptions at all, and if they are to be used, decide the degree and manner of their usage. At the system level, the architect must identify the exceptions that are possible and how they will be addressed.

98.1 Do Not Raise Language-Defined Exceptions (EXU01)

Level → Required

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



**Performance
Security**

Remediation → Low

Verification Method → GNATcheck rule: Raising_Predefined_Exceptions (builtin rule)

98.1.1 Reference

[SEI-Java] ERR07-J

98.1.2 Description

In no case should the application explicitly raise a language-defined exception.

The Ada language-defined exceptions are raised implicitly in specific circumstances defined by the language standard. Explicitly raising these exceptions would be confusing to application developers. The potential for confusion increases as the exception is propagated up the dynamic call chain, away from the point of the **raise** statement, because this increases the number of paths and thus corresponding language-defined checks that could have been the cause.

98.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A

98.1.4 Applicable Common Weakness Enumeration

- **CWE-397 - Declaration of Throws for Generic Exception**⁵¹⁰

98.1.5 Noncompliant Code Example

⁵¹⁰ <https://cwe.mitre.org/data/definitions/397.html>

```

procedure Noncompliant (X : in out Integer) is
begin
  if X < Integer'Last / 2
  then
    X := X * 2;
  else
    raise Constraint_Error;
  end if;
end Noncompliant;

```

98.1.6 Compliant Code Example

```

procedure Compliant (X : in out Integer) is
begin
  if X < Integer'Last / 2
  then
    X := X * 2;
  else
    raise Math_Overflow;
  end if;
end Compliant;

```

98.1.7 Notes

N/A

98.2 No Unhandled Application-Defined Exceptions (EXU02)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: Unhandled_Exceptions (supplied with document)

98.2.1 Reference

N/A

98.2.2 Description

All application-defined exceptions must have at least one corresponding handler that is applicable. Otherwise, if an exception is raised, undesirable behavior is possible. The term *applicable* means that there is no dynamic call chain that can reach the active exception which does not also include a handler that will be invoked for that exception, somewhere in that chain.

When an unhandled exception occurs in the sequence of statements of an application task and propagates to task's body, the task terminates abnormally. No *notification* of some sort is required or defined by the language, although some vendors' implementations may print out a log message or provide some other non-standard response. (Note that such a notification implies an external persistent environment, such as an operating system, that may not be present in all platforms.) The task failure does not affect any other tasks unless those other tasks attempt to communicate with it. In short, failure is silent.

Although the language-defined package `Ada.Task_Termination` can be used to provide a response using standard facilities, not all run-time libraries provide that package. For example, under the Ravenscar profile, application tasks are not intended to terminate, neither normally nor abnormally, and the language does not define what happens if they do. A run-time library for a memory-constrained target, especially a bare-metal target without an operating system, might not include any support for task termination when the tasking model is Ravenscar. The effects of task termination in that case are not defined by the language.

When an unhandled exception occurrence reaches the main subprogram and is not handled there, the exception occurrence is propagated to the environment task, which then completes abnormally. Even if the main subprogram does handle the exception, the environment task still completes (normally in that case).

When the environment task completes (normally or abnormally) it waits for the completion of dependent application tasks, if any. Those dependent tasks continue executing normally, i.e., they do not complete as a result of the environment task completion. Alternatively, however, instead of waiting for them, the implementation has permission to abort the dependent application tasks, per [Ada Reference Manual: 10.2 \(30\) Program Execution](#)⁵¹¹. The resulting application-specific effect is undefined.

Finally, whether the environment task waited for the dependent tasks or aborted them, the semantics of further execution beyond that point are undefined. There is no concept of a calling environment beyond the environment task ([Ada Reference Manual: 10.2 \(30\) Program Execution](#)⁵¹²). In some systems there is no calling environment, such as bare-metal platforms with only an Ada run-time library and no operating system.

98.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.36 Ignored error status and unhandled exceptions [OYB]

98.2.4 Applicable Common Weakness Enumeration

- **CWE-248 - Uncaught Exception**⁵¹³

⁵¹¹ <http://www.ada-auth.org/standards/12rm/html/RM-10-2.html>

⁵¹² <http://www.ada-auth.org/standards/12rm/html/RM-10-2.html>

⁵¹³ <https://cwe.mitre.org/data/definitions/248.html>

98.2.5 Noncompliant Code Example

```

procedure Main is
begin
  if Argument_Count = 0 then
    raise Cli_Exception;
  else
    begin
      Start_Application (Argument (1));
    exception
      when Application_Exception =>
        Put_Line ("Application failed");
    end;
  end if;
end Main;

```

98.2.6 Compliant Code Example

```

procedure Main is
begin
  if Argument_Count = 0 then
    raise Cli_Exception;
  else
    begin
      Start_Application (Argument (1));
    exception
      when Application_Exception =>
        Put_Line ("Application failed");
    end;
  end if;
exception
  when Cli_Exception =>
    Put_Line ("Failure");
end Main;

```

98.2.7 Notes

SPARK can prove that no exception will be raised (or fail to prove it and indicate the failure).

98.3 No Exception Propagation Beyond Name Visibility (EXU03)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

Security

Remediation → Low

Verification Method → GNATcheck rule: Non_Visible_Exceptions (builtin rule)

98.3.1 Reference

RPP05

98.3.2 Description

An active exception can be propagated dynamically past the point where the name of the exception is visible (the scope of the declaration). The exception can only be handled via **others** past that point. That situation prevents handling the exception specifically, and violates RPP05.

98.3.3 Applicable Vulnerability within ISO TR 24772-2

N/A

98.3.4 Applicable Common Weakness Enumeration

- **CWE-248 - Uncaught Exception**⁵¹⁴

98.3.5 Noncompliant Code Example

```
procedure Noncompliant (Param : in out Integer) is
  Noncompliant_Exception : exception;
begin
  Param := Param * Param;
exception
  when others =>
    raise Noncompliant_Exception;
end Noncompliant;
```

As a result the exception name cannot be referenced outside the body:

```
procedure Bad_Call (Param : in out Integer) is
begin
  Noncompliant (Param);
exception
  when Noncompliant_Exception => -- compile error
    null;
end Bad_Call;
```

98.3.6 Compliant Code Example

```
Compliant_Exception : exception;
procedure Compliant (Param : in out Integer) is
begin
  Param := Param * Param;
```

(continues on next page)

⁵¹⁴ <https://cwe.mitre.org/data/definitions/248.html>

(continued from previous page)

```

exception
  when others =>
    raise Compliant_Exception;
end Compliant;

procedure Good_Call (Param : in out Integer) is
begin
  Compliant (Param);
exception
  when Compliant_Exception =>
    null;
end Good_Call;

```

98.3.7 Notes

N/A

98.4 Prove Absence of Run-time Exceptions (EXU04)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

Security

Remediation → Low

Verification Method → Compiler restrictions

98.4.1 Reference

MISRA C Rule 1.3: "There shall be no occurrence of undefined or critical unspecified behaviour."

98.4.2 Description

In many high-integrity systems the possible responses to an exception are limited or nonexistent. In these cases the only approach is to prove exceptions cannot occur in the first place. Additionally, the cost of proving exceptions cannot happen may be less than the cost of analyzing code in which they are allowed to be raised.

The restriction `No_Exceptions` can be used with `pragma Restrictions` to enforce this approach. Specifically, the restriction ensures that `raise` statements and exception handlers

do not appear in the source code and that language-defined checks are not emitted by the compiler. However, a run-time check performed automatically by the hardware is permitted because it typically cannot be prevented. An example of such a check would be traps on invalid addresses. If a hardware check fails, or if an omitted language-defined check would have failed, execution is unpredictable. As a result, enforcement with the restriction is not ideal. However, proof of the absence of run-time errors is possible using the SPARK subset of Ada.

98.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

98.4.4 Applicable Common Weakness Enumeration

- **CWE-248 - Uncaught Exception**⁵¹⁵

98.4.5 Noncompliant Code Example

N/A

98.4.6 Compliant Code Example

N/A

98.4.7 Notes

This restriction is detected by SPARK, in which any statements explicitly raising an exception must be proven unreachable (or proof fails and the failure is indicated), and any possibility of run-time exception should be proved not to happen.

⁵¹⁵ <https://cwe.mitre.org/data/definitions/248.html>

OBJECT-ORIENTED PROGRAMMING (OOP)

Goal

Maintainability



Reliability



Portability
Performance
Security



Description

Have a plan for selecting the OOP facilities of the language to use.

Rules

OOP01, OOP02, OOP03, OOP04, OOP05, OOP06, OOP07

There are many issues to consider when planning the use of Object Oriented features in a high-integrity application. Choices should be made based on the desired expressive power of the OO features and the required level of certification or safety case.

For example, the use of inheritance can provide abstraction and separation of concerns. However, the extensive use of inheritance, particularly in deep hierarchies, can lead to fragile code bases.

Similarly, when new types of entities are added, dynamic dispatching provides separation of the code that must change from the code that manipulates those types and need not be changed to handle new types. However, analysis of dynamic dispatching must consider every candidate object type and analyze the associated subprogram for appropriate behavior.

Therefore, the system architect has available a range of possibilities for the use of OOP constructs, with tool enforcement available for the selections. Note that full use of OOP, including dynamic dispatching, may not be unreasonable.

The following rules assume use of tagged types, a requirement for full OOP in Ada.

99.1 No Class-wide Constructs Policy (OOP01)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability

✓

Reliability

✓

Portability
Performance
Security

✓

Remediation → N/A

Verification Method → Compiler restrictions

Mutually Exclusive → OOP02

99.1.1 Reference

N/A

99.1.2 Description

In this approach, tagged types are allowed and type extension (inheritance) is allowed, but there are no class-wide constructs.

This restriction ensures there are no class-wide objects or formal parameters, nor access types designating class-wide types.

In this approach there are no possible dynamic dispatching calls because such calls can only occur when a class-wide value is passed as the parameter to a primitive operation of a tagged type.

99.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]

99.1.4 Applicable Common Weakness Enumeration

N/A

99.1.5 Noncompliant Code Example

```
X : Object'Class := Some_Object;
```

99.1.6 Compliant Code Example

```
X : Object := Some_Object;
```

99.1.7 Notes

The compiler will detect violations with the standard Ada restriction `No_Dispatch` applied.

99.2 Static Dispatching Only Policy (OOP02)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability

Performance

Security



Remediation → N/A

Verification Method → Compiler restrictions

Mutually Exclusive → OOP01

99.2.1 Reference

N/A

99.2.2 Description

In this approach, class-wide constructs are allowed, as well as tagged types and type extension (inheritance), but dynamic dispatching remains disallowed (i.e., as in OOP01).

This rule ensures there are no class-wide values passed as the parameter to a primitive operation of a tagged type, hence there are no dynamically dispatched calls.

Note that this rule should not be applied without due consideration.

99.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]

99.2.4 Applicable Common Weakness Enumeration

N/A

99.2.5 Noncompliant Code Example

```
Some_Primitive (Object'Class (X));
```

99.2.6 Compliant Code Example

```
Some_Primitive (X);
```

99.2.7 Notes

N/A

99.3 Limit Inheritance Hierarchy Depth (OOP03)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability

Performance

Security



Remediation → High

Verification Method → GNATcheck rule: Deep_Inheritance_Hierarchies:2 (builtin rule)

99.3.1 Reference

[AdaOOP2016] section 5.1

99.3.2 Description

A class inheritance hierarchy consists of a set of types related by inheritance. Each class, other than the root class, is a subclass of other classes, and each, except for "leaf" nodes, is a base class for those that are derived from it.

Improperly designed inheritance hierarchies complicate system maintenance and increase the effort in safety certification, in any programming language.

A common characteristic of problematic hierarchies is "excessive" depth, in which a given class is a subclass of many other classes. Depth can be a problem because a change to a class likely requires inspection, modification, recompilation, and retesting/reverification of all classes below it in the hierarchy. The extent of that effect increases as we approach the root class. This rippling effect is known as the *fragile base class* problem. Clearly, the greater the depth the more subclasses there are to be potentially affected. In addition, note that a change to one class may cause a cascade of other secondary changes to subclasses, so the effect is often not limited to a single change made to all the subclasses in question.

Deep inheritance hierarchies also contribute to complexity, rather than lessening it, by requiring the reader to understand multiple superclasses in order to understand the behavior of a given subclass.

99.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.41 Inheritance [RIP]

99.3.4 Applicable Common Weakness Enumeration

- **CWE-1074 - Class with Excessively Deep Inheritance**⁵¹⁶
- **CWE-1086 - Class with Excessive Number of Child Classes**⁵¹⁷

99.3.5 Noncompliant Code Example

The threshold for "too deep" is inexact, but beyond around 4 or 5 levels the complexity accelerates rapidly.

```
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Trapezoid_T is new Quadrilateral_T with private;
type Parallelogram_T is new Trapezoid_T with private;
type Rectangle_T is new Parallelogram_T with private;
type Square_T is new Rectangle_T with private;
```

99.3.6 Compliant Code Example

```
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Rectangle_T is new Quadrilateral_T with private;
type Square_T is new Rectangle_T with private;
```

99.3.7 Notes

N/A

99.4 Limit Statically-Dispatched Calls to Primitive Operations (OOP04)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

⁵¹⁶ <https://cwe.mitre.org/data/definitions/1074.html>

⁵¹⁷ <https://cwe.mitre.org/data/definitions/1086.html>

Reliability



Portability



Performance
Security

Remediation → Medium (easy fix, but a difficult to detect bug)

Verification Method → GNATcheck rule: `Direct_Calls_To_Primitives` (builtin rule)

99.4.1 Reference

N/A

99.4.2 Description

This rule applies only to tagged types, when visibly tagged at the point of a call from one primitive to another of that same type.

By default, subprogram calls are statically dispatched. Dynamic dispatching only occurs when a class-wide value is passed to a primitive operation of a specific type. Forcing an otherwise optional dynamic dispatching call in this case is known as *redispatching*.

When one primitive operation of a given tagged type invokes another distinct primitive operation of that same type, use redispatching so that an overriding version of that other primitive will be invoked if it exists. Otherwise an existing overridden version would not be invoked, which is very likely an error.

This rule does not apply to the common case in which an overriding of a primitive operation calls the "parent" type's version of the overridden operation. Such calls occur in the overridden body when the new version is not replacing, but rather, is augmenting the parent type's version. In this case the new version must do whatever the parent version did, and can then add functionality specific to the new type.

By default, this rule applies to another common case in which static calls from one primitive operation to another make sense. Specifically, *constructors* are often implemented in Ada as functions that create a new value of the tagged type. As constructors, these functions are type-specific. They must call the primitive operations of the type being created, not operations that may be overridden for some type later derived from it. (Note that there is a GNATcheck rule parameter to not flag this case.)

Typically constructor functions only have the tagged type as the result type, not as the type for formal parameters, if any, because actual parameters of the tagged type would themselves likely require construction. This specific usage is the case ignored by the GNATcheck rule parameter.

Note that constructors implemented as procedures also call primitive operations of the specific type, for the same reasons as constructor functions. This usage is allowed by this rule and does not require the GNATcheck parameter. (The difference between function and procedure constructors is that these procedures will have a formal parameter of the tagged type, of mode **out**.)

99.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispatching [PPH]
- 6.44 Polymorphic variables [BKK]

99.4.4 Applicable Common Weakness Enumeration

N/A

99.4.5 Noncompliant Code Example

Class constructs

```
package Root is
  type Root_T is tagged null record;
  procedure Noncompliant (X : in out Root_T) is null;
  procedure Compliant (X : in out Root_T) is null;
  procedure Other_Prim (X : in out Root_T) is null;
end Root;

package Child is
  use Root;
  type Child_T is new Root_T with null record;
  procedure Noncompliant (X : in out Child_T);
  procedure Compliant (X : in out Child_T);
  procedure Other_Prim (X : in out Child_T);
end Child;

procedure Not_A_Primitive (X : in out Child.Child_T) is null;
```

Noncompliant Code

```
procedure Noncompliant (X : in out Child_T) is
begin
  Other_Prim (Root_T (X));
  Other_Prim (X);
end Noncompliant;
```

99.4.6 Compliant Code Example

```
procedure Compliant (X : in out Child_T) is
begin
  Compliant (Root_T (X)); -- constructor style is OK
  Not_A_Primitive (X);
end Compliant;
```

99.4.7 Notes

N/A

99.5 Use Explicit Overriding Annotations (OOP05)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

Security

Remediation → Low

Verification Method → GNATcheck rule: Style_Checks:0 (builtin rule)

99.5.1 Reference

[AdaOOP2016] section 4.3

99.5.2 Description

The declaration of a primitive operation that overrides an inherited operation must include an explicit **overriding** annotation.

The semantics of inheritance in mainstream object-oriented languages may result in two kinds of programming errors: 1) intending, but failing, to override an inherited subprogram, and 2) intending not to override an inherited subprogram, but doing so anyway. Because an overridden subprogram may perform subclass-specific safety or security checks, the invocation of the parent subprogram on a subclass instance can introduce a vulnerability.

The first issue (intending but failing to override) typically occurs when the subprogram name is misspelled. In this case a new or overloaded subprogram is actually declared.

The second issue (unintended overriding) can arise when a new subprogram is added to a parent type in an existing inheritance hierarchy. The new subprogram happens to cause one or more inherited subprograms below it to override the new superclass version. This mistake typically happens during program maintenance.

In Ada, much like other modern languages, one can annotate a subprogram declaration (and body) with an indication that the subprogram is an overriding of an inherited version. This is done with the **overriding** reserved word preceding the subprogram specification.

Similarly, in Ada one can explicitly indicate that a subprogram is not an overriding. To do so, the programmer includes the reserved words **not overriding** immediately prior to the subprogram specification.

Of course, incorrect marking errors are flagged by the compiler. If a subprogram is explicitly marked as overriding but is not actually overriding, the compiler will reject the code. Likewise, if a primitive subprogram is explicitly marked as not overriding, but actually is overriding, the compiler will reject the code.

However, most subprograms are not overriding so it would be a heavy burden on the programmer to make them explicitly indicate that fact. That's not to mention the relatively heavy syntax required.

In addition, both annotations are optional for the sake of compatibility with prior versions of the language. Therefore, a subprogram without either annotation might or might not be overriding. A legal program could contain some explicitly annotated subprograms and some that are not annotated at all. But because the compiler will reject explicit annotations that are incorrect, all we require is that one of the two cases be explicitly indicated for all such subprograms. Any unannotated subprograms not flagged as errors are then necessarily not that case, they must be the other one.

Since overriding is less common and involves slightly less syntax to annotate, the guideline requires explicit annotations only for overriding subprograms. It follows that any subprograms not flagged as errors by the compiler are not overriding, so they need not be marked explicitly as such.

This guideline is implemented by compiler switches, or alternatively, by a GNATcheck rule (specified below the table). With this guideline applied and enforced, the two inheritance errors described in the introduction cannot happen.

Note that the compiler switches will also require the explicit overriding indicator when overriding a language-defined operator. The switches also apply to inherited primitive subprograms for non-tagged types.

99.5.3 Applicable Vulnerability within ISO TR 24772-2

- 6.34 Subprogram signature mismatch [OTR]
- 6.41 Inheritance [RIP]

99.5.4 Applicable Common Weakness Enumeration

- **CWE-685 - Function Call With Incorrect Number of Arguments**⁵¹⁸

99.5.5 Noncompliant Code Example

```
type Root_T is tagged null record;
procedure Primitive (X : in out Root_T) is null;
```

```
type Noncompliant_Child_T is new Root_T with null record;
procedure Primitive (X : in out Noncompliant_Child_T) is null;
```

99.5.6 Compliant Code Example

```
type Compliant_Child_T is new Root_T with null record;
overriding procedure Primitive (X : in out Compliant_Child_T) is null;
```

99.5.7 Notes

This rule requires the GNAT compiler switches `-gnaty0` and `-gnatwe` in order for the compiler to flag missing overriding annotations as errors. The first causes the compiler to generate the warnings, and the second causes those warnings to be treated as errors.

99.6 Use Class-wide Pre/Post Contracts (OOP06)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

⁵¹⁸ <https://cwe.mitre.org/data/definitions/685.html>

Maintainability

✓

Reliability

✓

Portability

Performance

Security

✓

Remediation → Low

Verification Method → GNATcheck rule: Specific_Pre_Post (builtin rule)

99.6.1 Reference

[AdaOOP2016] section 6.1.4

SPARK User's Guide, section 7.5.2⁵¹⁹

99.6.2 Description

For primitive operations of tagged types, use only class-wide pre/post contracts, if any.

The class-wide form of precondition and postcondition expresses conditions that are intended to apply to any version of the subprogram. Therefore, when a subprogram is derived as part of inheritance, only the class-wide form of those contracts is inherited from the parent subprogram, if any are defined. As a result, it only makes sense to use the class-wide form in this situation.

(The same semantics and recommendation applies to type invariants.)

Note: this approach will be required for OOP07 (Ensure Local Type Consistency).

99.6.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

99.6.4 Applicable Common Weakness Enumeration

N/A

99.6.5 Noncompliant Code Example

```
type Root_T is tagged null record;
procedure Set_Name (X : Root_T;
                   Name : String)
  with Pre => Name'length > 0;
function Get_Name (X : Root_T) return String
  with Post => Get_Name'result'length > 0;
```

99.6.6 Compliant Code Example

⁵¹⁹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_object_oriented_contracts.html#writing-contracts-on-dispatching-subprograms

```

type Root_T is tagged null record;
procedure Set_Name (X      : Root_T;
                   Name : String)
  with Pre'class => Name'length > 0;
function Get_Name (X : Root_T) return String
  with Post'class => Get_Name'result'length > 0;

```

99.6.7 Notes

N/A

99.7 Ensure Local Type Consistency (OOP07)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability
Performance
Security

✓

Remediation → N/A

Verification Method → Software test

99.7.1 Reference

[AdaOOP2016] See section 4.2.

GNAT User's Guide, section 5.10.11⁵²⁰

99.7.2 Description

Either:

- Formally verify local type consistency, or
- Ensure that each tagged type passes all the tests of all the parent types which it can replace.

Rationale:

One of the fundamental benefits of OOP is the ability to manipulate objects in a class inheritance hierarchy without "knowing" at compile-time the specific classes of the objects being manipulated. By *manipulate* we mean invoking the primitive operations, the *methods* defined by the classes.

⁵²⁰ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html

We will use the words *class* and *type* interchangeably, because classes are composed in Ada and SPARK using a combination of building blocks, especially type declarations. However, we will use the term *subclass* rather than *subtype* because the latter has a specific meaning in Ada and SPARK that is unrelated to OOP.

The objects whose operations are being invoked can be of types anywhere in the inheritance tree, from the root down to the bottom. The location, i.e., the specific type, is transparent to the manipulating code. This type transparency is possible because the invoked operations are dynamically dispatched at run-time, rather than statically dispatched at compile-time.

Typically, the code manipulating the objects does so in terms of superclasses closer to the root of the inheritance tree. Doing so increases generality because it increases the number of potential subclasses that can be manipulated. The actual superclass chosen will depend on the operations required by the manipulation. In Ada and SPARK, subclasses can add operations but can never remove them, so more operations are found as we move down from the root. That is the nature of specialization. Note that the guarantee of an invoked operations' existence is essential for languages used in this domain.

However, for this transparent manipulation to be functionally correct — to accomplish what the caller intends — the primitive operations of subclasses must be functionally indistinguishable from those of the superclasses. That doesn't mean the subclasses cannot make changes. Indeed, the entire point of subclasses is to make changes. In particular, functional changes can be either introduction of new operations, or overridings of inherited operations. It is these overridings that must be functionally transparent to the manipulating code. (Of course, for an inherited operation that is not overridden, the functionality is inherited as-is, and is thus transparent trivially.)

The concept of functional transparency was introduced, albeit with different terminology, by Liskov and Wing in 1994 [[LiskovWing1994](#)] and is, therefore, known as the Liskov Substitution Principle, or LSP. The *substitution* in LSP means that a subclass must be substitutable for its superclass, i.e., a subclass instance should be usable whenever a superclass instance is required.

Unfortunately, requirements-based testing will not detect violations of LSP because unit-level requirements do not concern themselves with superclass substitutability.

However, the OO supplement of DO-178C [[DO178C](#)] offers solutions, two of which are in fact the options recommended by this guideline.

Specifically, the supplement suggests adding a specific verification activity it defines as Local Type Consistency Verification. This activity ensures LSP is respected and, in so doing, addresses the vulnerability.

Verification can be accomplished statically with formal methods in SPARK, or dynamically via a modified form of testing.

For the static approach, type consistency is verified by examining the properties of the overriding operation's preconditions and postconditions. These are the properties required by Design by Contract, as codified by Bertrand Meyer [[Meyer1997](#)]. Specifically, an overridden primitive may only replace the precondition with one weaker than that of the parent version, and may only replace the postcondition with one stronger. In essence, relative to the parent version, an overridden operation can only require the same or less, and provide the same or more. Satisfying that requirement is sufficient to ensure functional transparency because the manipulating code only "knows" the contracts of the class it manipulates, i.e., the view presented by the type, which may very well be a superclass of the one actually invoked.

In Ada and SPARK, the class-wide form of preconditions and postconditions are inherited by overridden primitive operations of tagged types. The inherited precondition and that of the overriding (if any) are combined into a conjunction. All must hold, otherwise the precondition fails. Likewise, the inherited postcondition is combined with the overriding postcondition into a disjunction. At least one of them must hold. In Ada these are tested

at run-time. In SPARK, they are verified statically (or not, in which case proof fails and an error is indicated).

To verify substitutability via testing, all the tests for all superclass types are applied to objects of the given subclass type. If all the parent tests pass, this provides a high degree of confidence that objects of the new tagged type can properly substitute for parent type objects. Note that static proof of consistency provides an even higher degree of confidence.

99.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispaching [PPH]
- 6.44 Polymorphic variables [BKK]

99.7.4 Applicable Common Weakness Enumeration

N/A

99.7.5 Noncompliant Code Example

```
package P is
  pragma Elaborate_Body;
  type Rectangle is tagged private;
  procedure Set_Width (This : in out Rectangle;
                      Value : Positive)
  with
    Post => Width (This) = Value and
            Height (This) = Height (This'Old);

  function Width (This : Rectangle) return Positive;

  procedure Set_Height (This : in out Rectangle;
                       Value : Positive)
  with
    Post => Height (This) = Value and
            Width (This) = Width (This'Old);

  function Height (This : Rectangle) return Positive;
private
  ...
end P;
```

The postcondition for `Set_Width` states that the `Height` is not changed. Likewise, for `Set_Height`, the postcondition asserts that the `Width` is not changed. However, these postconditions are not class-wide so they are not inherited by subclasses.

Now, in a subclass `Square`, the operations are overridden so that setting the width also sets the height to the same value, and vice versa. Thus the overridden operations do not maintain type consistency, but this fact is neither detected at run-time, nor could SPARK verify it statically (and SPARK is not used at all in these versions of the packages).

```
with P; use P;
package Q is
  pragma Elaborate_Body;
  type Square is new Rectangle with private;

  overriding
  procedure Set_Width (This : in out Square;
```

(continues on next page)

(continued from previous page)

```

                                Value : Positive)
  with
    Post => Width (This) = Height (This);

  overriding
    procedure Set_Height (This : in out Square;
                          Value : Positive)
  with
    Post => Width (This) = Height (This);

private
  ...
end Q;

```

99.7.6 Compliant Code Example

```

package P with SPARK_Mode is
  pragma Elaborate_Body;
  type Rectangle is tagged private;

  procedure Set_Width (This : in out Rectangle;
                      Value : Positive)
  with
    Post'Class => Width (This) = Value and
                 Height (This) = Height (This'Old);

  function Width (This : Rectangle) return Positive;

  procedure Set_Height (This : in out Rectangle;
                       Value : Positive)
  with
    Post'Class => Height (This) = Value and
                 Width (This) = Width (This'Old);

  function Height (This : Rectangle) return Positive;

private
  ...
end P;

```

Now the postconditions are class-wide so they are inherited by subclasses. In the subclass Square, the postconditions will not hold at run-time. Likewise, SPARK can now prove that type consistency is not verified because the postconditions are weaker than those inherited:

```

with P; use P;
package Q with SPARK_Mode is
  pragma Elaborate_Body;
  type Square is new Rectangle with private;

  overriding
    procedure Set_Width (This : in out Square;
                        Value : Positive)
  with
    Post'Class => Width (This) = Height (This);

  overriding
    procedure Set_Height (This : in out Square;
                         Value : Positive)
  with
    Post'Class => Width (This) = Height (This);

```

(continues on next page)

(continued from previous page)

```
private
  type Square is new Rectangle with null record;
end Q;
```

99.7.7 Notes

Verification can be achieved dynamically with the GNATtest tool, using the `--validate-type-extensions` switch. SPARK enforces this rule.

SOFTWARE ENGINEERING (SWE)

Goal

Maintainability



Reliability



Portability



**Performance
Security**



Description

These rules promote "best practices" for software development.

Rules

SWE01, SWE02, SWE03, SWE04

100.1 Use SPARK Extensively (SWE01)

Level → Advisory

Category

Safety



Cyber



Goal

Maintainability



Reliability



Portability



**Performance
Security**



Remediation → High, as retrofit can be extensive

Verification Method → Compiler restrictions

100.1.1 Reference

SPARK User's Guide, section 8: "Applying SPARK in Practice"⁵²¹

100.1.2 Description

SPARK has proven itself highly effective, both in terms of low defects, low development costs, and high productivity. The guideline advises extensive use of SPARK, especially for the sake of formally proving the most critical parts of the source code. The rest of the code can be in SPARK as well, even if formal proof is not intended, with some parts in Ada when features outside the SPARK subset are essential.

100.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A

100.1.4 Applicable Common Weakness Enumeration

- **CWE-670 - Always-Incorrect Control Flow Implementation**⁵²²
- **CWE-754 - Improper Check for Unusual or Exceptional Conditions**⁵²³

100.1.5 Noncompliant Code Example

Any code outside the (very large) SPARK subset is flagged by the compiler.

100.1.6 Compliant Code Example

N/A

100.1.7 Notes

Violations are detected by the SPARK toolset.

100.2 Enable Optional Warnings and Treat As Errors (SWE02)

Level → Required

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

⁵²¹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/usage_scenarios.html

⁵²² <https://cwe.mitre.org/data/definitions/670.html>

⁵²³ <https://cwe.mitre.org/data/definitions/754.html>

Portability
Performance
Security



Remediation → Low

Verification Method → Compiler restrictions

100.2.1 Reference

Power of 10 rule #10: "All code must be compiled, from the first day of development, with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings."

100.2.2 Description

The Ada compiler does a degree of static analysis itself, and generates many warnings when they are enabled. These warnings likely indicate very real problems so they should be examined and addressed, either by changing the code or disabling the warning for the specific occurrence flagged in the source code.

To ensure that warnings are examined and addressed one way or the other, the compiler must be configured to treat warnings as errors, i.e., preventing object code generation.

Note that warnings will occasionally be given for code usage that is intentional. In those cases the warnings should be disabled by using `pragma Warnings` with the parameter `Off`, and a string indicating the error message to be disabled. In other cases, a different mechanism might be appropriate, such as `aspect (or pragma) Unreferenced`.

100.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.18 Dead Store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.22 Initialization of variables [LAV]

100.2.4 Applicable Common Weakness Enumeration

- **CWE-1127 - Compilation with Insufficient Warnings or Errors**⁵²⁴

100.2.5 Noncompliant Code Example

```
procedure P (This : Obj) is
begin
  ... code not referencing This
end P;
```

The formal parameter controls dispatching for the sake of selecting the subprogram to be called but does not participate in the implementation of the body.

⁵²⁴ <https://cwe.mitre.org/data/definitions/1127.html>

100.2.6 Compliant Code Example

```
procedure P (This : Obj) is
  pragma Unreferenced (This);
begin
  ... code not referencing This
end P;
```

The compiler will no longer issue a warning that the formal parameter `This` is not referenced. Of course, if that changes and `This` becomes referenced, the compiler will flag the `pragma`.

100.2.7 Notes

This rule can be applied via the GNAT -gnatwae compiler switch, which both enables warnings and treats them as errors. Note that the switch enables almost all optional warnings, but not all. Some optional warnings correspond to very specific circumstances, and would otherwise generate too much noise for their value.

100.3 Use a Static Analysis Tool Extensively (SWE03)

Level → Mandatory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability

✓

Performance

✓

Security

✓

Remediation → High

Verification Method → Static analysis tools

100.3.1 Reference

Power of 10 rule #10: "All code must also be checked daily with at least one, but preferably more than one, strong static source code analyzer and should pass all analyses with zero warnings."

100.3.2 Description

If not using SPARK for regular development, use a static analyzer, such as CodePeer, extensively. No warnings or errors should remain unresolved at the given level adopted for analysis (which can be selected to adjust the false positive ratio).

Specifically, any code checked into the configuration management system must be checked by the analyzer and be error-free prior to check-in. Similarly, each nightly build should produce a CodePeer baseline for the project.

100.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.6 Conversion errors [FLC]
- 6.18 Dead store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.24 Side-effects and order of evaluation [SAM]
- 6.25 Likely incorrect expression [KOA]

100.3.4 Applicable Common Weakness Enumeration

N/A

100.3.5 Noncompliant Code Example

N/A

100.3.6 Compliant Code Example

N/A

100.3.7 Notes

CodePeer is the recommended static analyzer. Note that CodePeer can detect GNATcheck rule violations (via the `--gnatcheck` CodePeer switch and a rules file).

100.4 Hide Implementation Artifacts (SWE04)

Level → Advisory

Category

Safety

✓

Cyber

✓

Goal

Maintainability

✓

Reliability

✓

Portability
Performance
Security



Remediation → High, as retrofit can be extensive

Verification Method → GNATcheck rule: Visible_Components (builtin rule)

100.4.1 Reference

MISRA C Rule 8.7: "Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."

100.4.2 Description

Do not make implementation artifacts compile-time visible to clients. Only make available those declarations that define the abstraction presented to clients by the component. In other words, define Abstract Data Types and use the language to enforce the abstraction. This is a fundamental Object-Oriented Design principle.

This guideline minimizes client dependencies and thus allows the maximum flexibility for changes in the underlying implementation. It minimizes the editing changes required for client code when implementation changes are made.

This guideline also limits the region of code required to find any bugs to the package and child packages, if any, defining the abstraction.

This guideline is to be followed extensively as the design default for components. Once the application code size becomes non-trivial, the cost of retrofit is extremely high.

100.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

100.4.4 Applicable Common Weakness Enumeration

- **CWE-496 - Public Data Assigned to Private Array-Typed Field**⁵²⁵
- **CWE-1061 - Insufficient Encapsulation**⁵²⁶

100.4.5 Noncompliant Code Example

```
package Noncompliant is
  type Content_T is array (Capacity_T range <>) of Integer;
  type Stack_T (Capacity : Capacity_T) is tagged record
    Content : Content_T (1 .. Capacity);
    Top      : Capacity_T := 0;
  end record;
  procedure Push
    (Stack : in out Stack_T;
     Item   : Integer);
  procedure Pop
    (Stack : in out Stack_T;
     Item   : out Integer);
end Noncompliant;
```

⁵²⁵ <https://cwe.mitre.org/data/definitions/496.html>

⁵²⁶ <https://cwe.mitre.org/data/definitions/1061.html>

Note that both type `Content_T`, as well as the record type components of type `Stack_T`, are visible to clients. Client code may declare variables of type `Content_T` and may directly access and modify the record components. Bugs introduced via this access could be anywhere in the entire client codebase.

100.4.6 Compliant Code Example

```
package Compliant is
  type Stack_T (Capacity : Capacity_T) is tagged private;
  procedure Push
    (Stack : in out Stack_T;
     Item  : Integer);
  procedure Pop
    (Stack : in out Stack_T;
     Item  : out Integer);
private
  type Content_T is array (Capacity_T range <>) of Integer;
  type Stack_T (Capacity : Capacity_T) is tagged record
    Content : Content_T (1 .. Capacity);
    Top     : Capacity_T := 0;
  end record;
end Compliant;
```

Type `Content_T`, as well as the record type components of type `Stack_T`, are no longer visible to clients. Any bugs in the stack processing code must be in this package, or its child packages, if any.

100.4.7 Notes

The GNATcheck rule specified above is not exhaustive.

REFERENCES

- AdaCore. [SPARK 2014 User's Guide](http://docs.adacore.com/spark2014-docs/html/ug/index.html).⁵²⁹
- Adacore. [GNAT User's Guide for Native Platforms](http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html)⁵³⁰
- AdaCore. ["GNATstack User's Guide"](http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html)⁵³¹

⁵²⁹ <http://docs.adacore.com/spark2014-docs/html/ug/index.html>

⁵³⁰ http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html

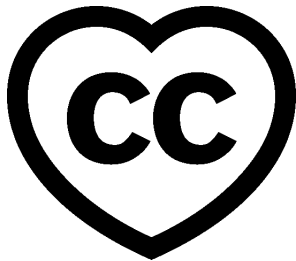
⁵³¹ http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html

Part X

What's New in Ada 2022

Copyright © 2022 – 2023, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)⁵³²



This course presents an overview of the new features of the latest Ada 2022 standard.

This document was written by Maxim Reznik and reviewed by Richard Kenner.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can [download from the learn website](https://learn.adacore.com/zip/learning-ada_code.zip)⁵³³. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/  
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;
5. Run the application (if a main procedure is available in the project).

⁵³² [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

⁵³³ https://learn.adacore.com/zip/learning-ada_code.zip

INTRODUCTION

This is a collection of short code examples demonstrating new features of the [Ada 2022 Standard](#)⁵³⁴ as they are implemented in GNAT Ada compiler.

To use some of these features, you may need to use a compiler command line switch or pragma. Compilers starting with [GNAT Community Edition 2021](#)⁵³⁵ or [GCC 11](#)⁵³⁶ use **pragma Ada_2022**; or the `-gnat2022` switch.

102.1 References

- Draft [Ada 2022 Standard](#)⁵³⁷

⁵³⁴ <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

⁵³⁵ <https://blog.adacore.com/gnat-community-2021-is-here>

⁵³⁶ <https://gcc.gnu.org/gcc-11/>

⁵³⁷ <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

'IMAGE ATTRIBUTE FOR ANY TYPE

Note

Attribute `'Image` for any type is supported by

- GNAT Community Edition 2020 and latter
- GCC 11

103.1 'Image attribute for a value

Since the publication of the [Technical Corrigendum 1⁵³⁸](#) in February 2016, the `'Image` attribute can now be applied to a value. So instead of `My_Type'Image (Value)`, you can just write `Value'Image`, as long as the `Value` is a [name⁵³⁹](#). These two statements are equivalent:

```
Ada.Text_IO.Put_Line (Ada.Text_IO.Page_Length'Image);

Ada.Text_IO.Put_Line
  (Ada.Text_IO.Count'Image (Ada.Text_IO.Page_Length));
```

103.2 'Image attribute for any type

In Ada 2022, you can apply the `'Image` attribute to any type, including records, arrays, access types, and private types. Let's see how this works. We'll define array, record, and access types and corresponding objects and then convert these objects to strings and print them:

Listing 2460: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4   type Vector is array (Positive range <>) of Integer;
5
6   V1 : aliased Vector := [1, 2, 3];
7
8   type Text_Position is record
9     Line, Column : Positive;
10  end record;
11
```

(continues on next page)

⁵³⁸ <https://reznikmm.github.io/ada-auth/rm-4-NC/RM-0-1.html>

⁵³⁹ <https://reznikmm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091>

(continued from previous page)

```
12  Pos : constant Text_Position := (Line => 10, Column => 3);
13
14  type Vector_Access is access all Vector;
15
16  V1_Ptr : constant Vector_Access := V1'Access;
17
18  begin
19      Ada.Text_IO.Put_Line (V1'Image);
20      Ada.Text_IO.Put_Line (Pos'Image);
21      Ada.Text_IO.New_Line;
22      Ada.Text_IO.Put_Line (V1_Ptr'Image);
23  end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Image_Attribute
MD5: c58d3133c45d780305a2b5c88d159764

Runtime output

```
[ 1,  2,  3]
(LINE => 10,
 COLUMN => 3)
(access 7fff5c9fbf18)
```

```
$ gprbuild -q -P main.gpr
Build completed successfully.
$ ./main
[ 1,  2,  3]
(LINE => 10,
 COLUMN => 3)
(access 7fff64b23988)
```

Note the square brackets in the array image output. In Ada 2022, array aggregates could be written *this way* (page 2097)!

103.3 References

- ARM 4.10 Image Attributes⁵⁴⁰
- AI12-0020-1⁵⁴¹

⁵⁴⁰ <http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html>

⁵⁴¹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0020-1.txt>

REDEFINING THE 'IMAGE ATTRIBUTE

In Ada 2022, you can redefine 'Image attribute for your type, though the syntax to do this has been changed several times. Let's see how it works in GNAT Community 2021.

Note

Redefining attribute 'Image is supported by

- GNAT Community Edition 2021 (using Text_Buffers)
- GNAT Community Edition 2020 (using Text_Output.Utils)
- GCC 11 (using Text_Output.Utils)

In our example, let's redefine the 'Image attribute for a location in source code. To do this, we provide a new Put_Image aspect for the type:

Listing 2461: main.adb

```
1 with Ada.Text_IO;
2 with Ada.Strings.Text_Buffers;
3
4 procedure Main is
5
6   type Source_Location is record
7     Line   : Positive;
8     Column : Positive;
9   end record
10   with Put_Image => My_Put_Image;
11
12   procedure My_Put_Image
13     (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
14      Value  : Source_Location);
15
16   procedure My_Put_Image
17     (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
18      Value  : Source_Location)
19   is
20     Line   : constant String := Value.Line'Image;
21     Column : constant String := Value.Column'Image;
22     Result : constant String :=
23       Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
24   begin
25     Output.Put (Result);
26   end My_Put_Image;
27
28   Line_10 : constant Source_Location := (Line => 10, Column => 1);
29
30 begin
```

(continues on next page)

(continued from previous page)

```
31   Ada.Text_IO.Put_Line (Line_10'Image);  
32 end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Image_Redefine
MD5: 4c77a3589eaba3a4c1b70200913ad6b8

Runtime output

10:1

104.1 What's the Root_Buffer_Type?

Let's see how it's defined in the `Ada.Strings.Text_Buffers` package.

```
type Root_Buffer_Type is abstract tagged limited private;  
  
procedure Put  
(Buffer : in out Root_Buffer_Type;  
 Item   : in String) is abstract;
```

In addition to `Put`, there are also `Wide_Put`, `Wide_Wide_Put`, `Put_UTF_8`, `Wide_Put_UTF_16`. And also `New_Line`, `Increase_Indent`, `Decrease_Indent`.

104.2 Outdated draft implementation

GNAT Community Edition 2020 and GCC 11 both provide a draft implementation that's incompatible with the Ada 2022 specification. For those versions, `My_Put_Image` looks like:

```
procedure My_Put_Image  
(Sink : in out Ada.Strings.Text_Output.Sink'Class;  
 Value : Source_Location)  
is  
  Line   : constant String := Value.Line'Image;  
  Column : constant String := Value.Column'Image;  
  Result : constant String :=  
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);  
begin  
  Ada.Strings.Text_Output.Utils.Put_UTF_8 (Sink, Result);  
end My_Put_Image;
```

104.3 References

- ARM 4.10 Image Attributes⁵⁴²
- AI12-0020-1⁵⁴³
- AI12-0384-2⁵⁴⁴

⁵⁴² <http://www.ada-auth.org/standards/22aarm/html/AA-4-10.html>

⁵⁴³ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT>

⁵⁴⁴ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT>

USER-DEFINED LITERALS

Note

User-defined literals are supported by

- GNAT Community Edition 2020
- GCC 11

In Ada 2022, you can define string, integer, or real literals for your types. The compiler will convert such literals to your type at run time using a function you provide. To do so, specify one or more new aspects:

- Integer_Literal
- Real_Literal
- String_Literal

For our example, let's define all three for a simple type and see how they work. For simplicity, we use a Wide_Wide_String component for the internal representation:

Listing 2462: main.adb

```
1 with Ada.Wide_Wide_Text_IO;
2 with Ada.Characters.Conversions;
3
4 procedure Main is
5
6   type My_Type (Length : Natural) is record
7     Value : Wide_Wide_String (1 .. Length);
8   end record
9   with String_Literal => From_String,
10      Real_Literal    => From_Real,
11      Integer_Literal => From_Integer;
12
13   function From_String (Value : Wide_Wide_String) return My_Type is
14     ((Length => Value'Length, Value => Value));
15
16   function From_Real (Value : String) return My_Type is
17     ((Length => Value'Length,
18      Value  => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
19
20   function From_Integer (Value : String) return My_Type renames From_Real;
21
22   procedure Print (Self : My_Type) is
23   begin
24     Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
25   end Print;
```

(continues on next page)

(continued from previous page)

```

26
27 begin
28   Print ("Test ""string""");
29   Print (123);
30   Print (16#DEAD_BEEF#);
31   Print (2.99_792_458e+8);
32 end Main;

```

Code block metadata

```

Project: Courses.Ada_2022_Whats_New.User_Defined_Literals
MD5: 755732418174c841bd88a8330cdc3e01

```

Runtime output

```

Test "string"
123
16#DEAD_BEEF#
2.99_792_458e+8

```

As you see, real and integer literals are converted to strings while preserving the formatting in the source code, while string literals are decoded: `From_String` is passed the specified string value. In all cases, the compiler translates these literals into function calls.

105.1 Turn Ada into JavaScript

Do you know that `'5'+3` in JavaScript is `53`?

```

> '5'+3
'53'

```

Now we can get the same result in Ada! But before we do, we need to define a custom `+` operator:

Listing 2463: main.adb

```

1 with Ada.Wide_Wide_Text_IO;
2 with Ada.Characters.Conversions;
3
4 procedure Main is
5
6   type My_Type (Length : Natural) is record
7     Value : Wide_Wide_String (1 .. Length);
8   end record
9   with String_Literal => From_String,
10    Real_Literal      => From_Real,
11    Integer_Literal   => From_Integer;
12
13   function "+" (Left, Right : My_Type) return My_Type is
14     (Left.Length + Right.Length, Left.Value & Right.Value);
15
16   function From_String (Value : Wide_Wide_String) return My_Type is
17     ((Length => Value'Length, Value => Value));
18
19   function From_Real (Value : String) return My_Type is
20     ((Length => Value'Length,
21      Value => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
22
23   function From_Integer (Value : String) return My_Type renames From_Real;

```

(continues on next page)

(continued from previous page)

```

24
25   procedure Print (Self : My_Type) is
26   begin
27       Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
28   end Print;
29
30   begin
31       Print ("5" + 3);
32   end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.User_Defined_Literals_JS
MD5: 2d6eaf2d1b5bf560a90d4c6e491a2495

Runtime output

53

Jokes aside, this feature is very useful. For example it allows a "native-looking API" for *big integers* (page 2117).

105.2 References

- ARM 4.2.1 User-Defined Literals⁵⁴⁵
- AI12-0249-1⁵⁴⁶
- AI12-0342-1⁵⁴⁷

⁵⁴⁵ <http://www.ada-auth.org/standards/22rm/html/RM-4-2-1.html>

⁵⁴⁶ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT>

⁵⁴⁷ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0342-1.TXT>

ADVANCED ARRAY AGGREGATES

Note

These array aggregates are supported by

- GNAT Community Edition 2020
- GCC 11

106.1 Square brackets

In Ada 2022, you can use square brackets in array aggregates. Using square brackets simplifies writing both empty aggregates and single-element aggregates. Consider this:

Listing 2464: show_square_brackets.ads

```
1 package Show_Square_Brackets is
2
3   type Integer_Array is array (Positive range <>) of Integer;
4
5   Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
6   New_Style_Empty : Integer_Array := [];
7
8   Old_Style_One_Item : Integer_Array := (1 => 5);
9   New_Style_One_Item : Integer_Array := [5];
10
11 end Show_Square_Brackets;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Square_Brackets
MD5: 2914b9435684a1eecd2567548969a3ed

Short summary for parentheses and brackets

- Record aggregates use parentheses
- *Container aggregates* (page 2101) use square brackets
- Array aggregates can use both square brackets and parentheses, but parentheses usage is obsolescent

106.2 Iterated Component Association

There is a new kind of component association:

```
Vector : Integer_Array := [for J in 1 .. 5 => J * 2];
```

This association starts with **for** keyword, just like a quantified expression. It declares an index parameter that you can use in the computation of a component.

Iterated component associations can nest and can be nested in another association (iterated or not). Here we use this to define a square matrix:

```
Matrix : array (1 .. 3, 1 .. 3) of Positive :=
  [for J in 1 .. 3 =>
    [for K in 1 .. 3 => J * 10 + K]];
```

Iterated component associations in this form provide both element indices and values, just like named component associations:

```
Data : Integer_Array (1 .. 5) :=
  [for J in 2 .. 3 => J, 5 => 5, others => 0];
```

Here Data contains (0, 2, 3, 0, 5), not (2, 3, 5, 0, 0).

Another form of iterated component association corresponds to a positional component association and provides just values, but no element indices:

```
Vector_2 : Integer_Array := [for X of Vector => X / 2];
```

You cannot mix these forms in a single aggregate.

It's interesting that such aggregates were originally proposed more than 25 years ago!

Complete code snippet:

Listing 2465: show_iterated_component_association.adb

```
1 with Ada.Text_IO;
2
3 procedure Show_Iterated_Component_Association is
4
5   type Integer_Array is array (Positive range <>) of Integer;
6
7   Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
8   New_Style_Empty : Integer_Array := [];
9
10  Old_Style_One_Item : Integer_Array := (1 => 5);
11  New_Style_One_Item : Integer_Array := [5];
12
13  Vector : constant Integer_Array := [for J in 1 .. 5 => J * 2];
14
15  Matrix : constant array (1 .. 3, 1 .. 3) of Positive :=
16    [for J in 1 .. 3 =>
17      [for K in 1 .. 3 => J * 10 + K]];
18
19  Data : constant Integer_Array (1 .. 5) :=
20    [for J in 2 .. 3 => J, 5 => 5, others => 0];
21
22  Vector_2 : constant Integer_Array := [for X of Vector => X / 2];
23 begin
24   Ada.Text_IO.Put_Line (Vector'Image);
25   Ada.Text_IO.Put_Line (Matrix'Image);
26   Ada.Text_IO.Put_Line (Data'Image);
```

(continues on next page)

(continued from previous page)

```

27   Ada.Text_IO.Put_Line (Vector_2'Image);
28 end Show_Iterated_Component_Association;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Iterated_Component_Association
MD5: 9bc822b59c2c423019917728aab75c69

Runtime output

```

[ 2,  4,  6,  8, 10]

[
  [ 11, 12, 13],
  [ 21, 22, 23],
  [ 31, 32, 33]]

[ 0,  2,  3,  0,  5]

[ 1,  2,  3,  4,  5]

```

106.3 References

- [ARM 4.3.3 Array Aggregates](#)⁵⁴⁸
- [AI12-0212-1](#)⁵⁴⁹
- [AI12-0306-1](#)⁵⁵⁰

⁵⁴⁸ <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html>

⁵⁴⁹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>

⁵⁵⁰ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT>

CONTAINER AGGREGATES

Note

Container aggregates are supported by

- GNAT Community Edition 2021
- GCC 11

Ada 2022 introduces container aggregates, which can be used to easily create values for vectors, lists, maps, and other aggregates. For containers such as maps, the aggregate must use named associations to provide keys and values. For other containers it uses positional associations. Only square brackets are allowed. Here's an example:

Listing 2466: main.adb

```
1 with Ada.Text_IO;
2 with Ada.Containers.Vectors;
3 with Ada.Containers.Ordered_Maps;
4
5 procedure Main is
6
7     package Int_Vectors is new Ada.Containers.Vectors
8         (Positive, Integer);
9
10    X : constant Int_Vectors.Vector := [1, 2, 3];
11
12    package Float_Maps is new Ada.Containers.Ordered_Maps
13        (Integer, Float);
14
15    Y : constant Float_Maps.Map := [-10 => 1.0, 0 => 2.5, 10 => 5.51];
16 begin
17     Ada.Text_IO.Put_Line (X'Image);
18     Ada.Text_IO.Put_Line (Y'Image);
19 end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_1
MD5: 317c252582f75f248a154bd843388aaa

Runtime output

```
[ 1,  2,  3]
[-10 =>  1.00000E+00,  0 =>  2.50000E+00,  10 =>  5.51000E+00]
```


At run time, the compiler creates an empty container and populates it with elements one by one. If you define a new container type, you can specify a new Aggregate aspect to enable container aggregates for your container and let the compiler know what subprograms to use to construct the aggregate:

Listing 2467: main.adb

```
1 procedure Main is
2
3   package JSON is
4     type JSON_Value is private
5       with Integer_Literal => To_JSON_Value;
6
7     function To_JSON_Value (Text : String) return JSON_Value;
8
9     type JSON_Array is private
10      with Aggregate => (Empty      => New_JSON_Array,
11                        Add_Unnamed => Append);
12
13     function New_JSON_Array return JSON_Array;
14
15     procedure Append
16       (Self : in out JSON_Array;
17        Value : JSON_Value) is null;
18
19   private
20     type JSON_Value is null record;
21     type JSON_Array is null record;
22
23     function To_JSON_Value (Text : String) return JSON_Value
24       is (null record);
25
26     function New_JSON_Array return JSON_Array is (null record);
27   end JSON;
28
29   List : JSON.JSON_Array := [1, 2, 3];
30   -----
31 begin
32   -- Equivalent old initialization code
33   List := JSON.New_JSON_Array;
34   JSON.Append (List, 1);
35   JSON.Append (List, 2);
36   JSON.Append (List, 3);
37 end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_2
MD5: d930702be4e6b4836afab082e8abe633

The equivalent for maps is:

Listing 2468: main.adb

```
1 procedure Main is
2
3   package JSON is
4     type JSON_Value is private
5       with Integer_Literal => To_JSON_Value;
6
7     function To_JSON_Value (Text : String) return JSON_Value;
8
```

(continues on next page)

(continued from previous page)

```

9      type JSON_Object is private
10         with Aggregate => (Empty      => New_JSON_Object,
11                             Add_Named => Insert);
12
13      function New_JSON_Object return JSON_Object;
14
15      procedure Insert
16         (Self : in out JSON_Object;
17          Key   : Wide_Wide_String;
18          Value : JSON_Value) is null;
19
20  private
21      type JSON_Value is null record;
22      type JSON_Object is null record;
23
24      function To_JSON_Value (Text : String) return JSON_Value
25         is (null record);
26
27      function New_JSON_Object return JSON_Object is (null record);
28  end JSON;
29
30  Object : JSON.JSON_Object := ["a" => 1, "b" => 2, "c" => 3];
31  -----
32  begin
33      -- Equivalent old initialization code
34      Object := JSON.New_JSON_Object;
35      JSON.Insert (Object, "a", 1);
36      JSON.Insert (Object, "b", 2);
37      JSON.Insert (Object, "c", 3);
38  end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_3
MD5: c39c651e3e2f48c515c87c4cc6b979e9

You can't specify both `Add_Named` and `Add_Unnamed` subprograms for the same type. This prevents you from defining `JSON_Value` with both array and object aggregates present. But we can define conversion functions for array and object and get code almost as dense as the same code in native JSON. For example:

Listing 2469: main.adb

```

1  procedure Main is
2
3      package JSON is
4          type JSON_Value is private
5              with Integer_Literal => To_Value, String_Literal => To_Value;
6
7          function To_Value (Text : String) return JSON_Value;
8          function To_Value (Text : Wide_Wide_String) return JSON_Value;
9
10         type JSON_Object is private
11             with Aggregate => (Empty      => New_JSON_Object,
12                                 Add_Named => Insert);
13
14         function New_JSON_Object return JSON_Object;
15
16         procedure Insert
17             (Self : in out JSON_Object;
18              Key   : Wide_Wide_String;

```

(continues on next page)

(continued from previous page)

```

19     Value : JSON_Value) is null;
20
21     function From_Object (Self : JSON_Object) return JSON_Value;
22
23     type JSON_Array is private
24       with Aggregate => (Empty      => New_JSON_Array,
25                          Add_Unnamed => Append);
26
27     function New_JSON_Array return JSON_Array;
28
29     procedure Append
30       (Self : in out JSON_Array;
31        Value : JSON_Value) is null;
32
33     function From_Array (Self : JSON_Array) return JSON_Value;
34
35 private
36     type JSON_Value is null record;
37     type JSON_Object is null record;
38     type JSON_Array is null record;
39
40     function To_Value (Text : String) return JSON_Value is
41       (null record);
42     function To_Value (Text : Wide_Wide_String) return JSON_Value is
43       (null record);
44     function New_JSON_Object return JSON_Object is
45       (null record);
46     function New_JSON_Array return JSON_Array is
47       (null record);
48     function From_Object (Self : JSON_Object) return JSON_Value is
49       (null record);
50     function From_Array (Self : JSON_Array) return JSON_Value is
51       (null record);
52 end JSON;
53
54 function "+" (X : JSON.JSON_Object) return JSON.JSON_Value
55   renames JSON.From_Object;
56 function "-" (X : JSON.JSON_Array) return JSON.JSON_Value
57   renames JSON.From_Array;
58
59 Offices : JSON.JSON_Array :=
60   [+["name" => "North American Office",
61      "phones" => -[1_877_787_4628,
62                   1_866_787_4232,
63                   1_212_620_7300],
64      "email" => "info@adacore.com"],
65   +["name" => "European Office",
66      "phones" => -[33_1_49_70_67_16,
67                   33_1_49_70_05_52],
68      "email" => "info@adacore.com"]];
69 -----
70 begin
71   -- Equivalent old initialization code is too long to print it here
72   null;
73 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Container_Aggregates_4
MD5: a2770961eab00ee297fffa280cd9a481

The Offices variable is supposed to contain this value:

```
[{"name" : "North American Office",  
  "phones": [18777874628,  
             18667874232,  
             12126207300],  
  "email" : "info@adacore.com"},  
{"name" : "European Office",  
  "phones": [33149706716,  
             33149700552],  
  "email" : "info@adacore.com"}]
```

107.1 References

- [ARM 4.3.5 Container Aggregates](#)⁵⁵¹
- [AI12-0212-1](#)⁵⁵²

⁵⁵¹ <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-5.html>

⁵⁵² <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>

DELTA AGGREGATES

Note

Delta aggregates are supported by

- GNAT Community Edition 2019
- GCC 9

Sometimes you need to create a copy of an object, but with a few modifications. Before Ada 2022, doing this involves a dummy object declaration or an aggregate with associations for each property. The dummy object approach doesn't work in contract aspects or when there are limited components. On the other hand, re-listing properties in an large aggregate can be very tedious and error-prone. So, in Ada 2022, you can use a *delta aggregate* instead.

108.1 Delta aggregate for records

The delta aggregate for a record type looks like this:

```
type Vector is record
  X, Y, Z : Float;
end record;

Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
```

The more components you have, the more you will like the delta aggregate.

108.2 Delta aggregate for arrays

You can also use delta aggregates for arrays to change elements, but not bounds. Moreover, it only works for one-dimensional arrays of non-limited components.

```
type Vector_3D is array (1 .. 3) of Float;

Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
```

You can use parentheses for array aggregates, but you can't use square brackets for record aggregates.

Here is the complete code snippet:

Listing 2470: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4
5     type Vector is record
6         X, Y, Z : Float;
7     end record;
8
9     Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
10    Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
11
12    type Vector_3D is array (1 .. 3) of Float;
13
14    Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
15    Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
16 begin
17     Ada.Text_IO.Put (Float'Image (Projection_1.X));
18     Ada.Text_IO.Put (Float'Image (Projection_1.Y));
19     Ada.Text_IO.Put (Float'Image (Projection_1.Z));
20     Ada.Text_IO.New_Line;
21     Ada.Text_IO.Put (Float'Image (Projection_2 (1)));
22     Ada.Text_IO.Put (Float'Image (Projection_2 (2)));
23     Ada.Text_IO.Put (Float'Image (Projection_2 (3)));
24     Ada.Text_IO.New_Line;
25 end Main;
```

108.3 References

- ARM 4.3.4 Delta Aggregates⁵⁵³
- AI12-0127-1⁵⁵⁴

⁵⁵³ <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html>

⁵⁵⁴ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0127-1.TXT>

TARGET NAME SYMBOL (@)

Note

Target name symbol is supported by

- GNAT Community Edition 2019
- GCC 9

Ada 2022 introduces a new symbol, @, which can only appear on the right hand side of an assignment statement. This symbol acts as the equivalent of the name on the left hand side of that assignment statement. It was introduced to avoid code duplication: instead of retyping a (potentially long) name, you can use @. This symbol denotes a constant, so you can't pass it into **[in] out** arguments of a subprogram.

As an example, let's calculate some statistics for My_Data array:

Listing 2471: statistics.ads

```
1 package Statistics is
2
3     type Statistic is record
4         Count : Natural := 0;
5         Total : Float := 0.0;
6     end record;
7
8     My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
9
10    Statistic_For_My_Data : Statistic;
11
12 end Statistics;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_Intro
MD5: 8ca75d894ed8ddb459aee0ce0c427d

To do this, we loop over My_Data elements:

Listing 2472: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4
5     type Statistic is record
6         Count : Natural := 0;
7         Total : Float := 0.0;
```

(continues on next page)

(continued from previous page)

```
8   end record;
9
10  My_Data : constant array (1 .. 5) of Float :=
11    [for J in 1 .. 5 => Float (J)];
12
13  Statistic_For_My_Data : Statistic;
14
15  begin
16    for Data of My_Data loop
17      Statistic_For_My_Data.Count := @ + 1;
18      Statistic_For_My_Data.Total := @ + Data;
19    end loop;
20
21    Ada.Text_IO.Put_Line (Statistic_For_My_Data'Image);
22  end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_2
MD5: b116e7a172d43547b227eb27fb994366

Runtime output

```
(COUNT => 5,
TOTAL => 1.50000E+01)
```

Each right hand side is evaluated only once, no matter how many @ symbols it contains. Let's verify this by introducing a function call that prints a line each time it's called:

Listing 2473: main.adb

```
1  with Ada.Text_IO;
2
3  procedure Main is
4
5    My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
6
7    function To_Index (Value : Positive) return Positive is
8    begin
9      Ada.Text_IO.Put_Line ("To_Index is called.");
10     return Value;
11   end To_Index;
12
13   begin
14     My_Data (To_Index (1)) := @ ** 2 - 3.0 * @;
15     Ada.Text_IO.Put_Line (My_Data'Image);
16   end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Assignment_Tagged_3
MD5: ed946d4e1b90df28fbd358d17033279d

Runtime output

To_Index is called.

```
[-2.00000E+00, 2.00000E+00, 3.00000E+00, 4.00000E+00, 5.00000E+00]
```

This use of @ may look a bit cryptic, but it's the best solution that was found. Unlike other languages (e.g., `sum += x;` in C), this approach lets you use @ an arbitrary number of times

within the right hand side of an assignment statement.

109.1 Alternatives

In C++, the previous statement could be written with a reference type (one line longer!):

```
auto& a = my_data[to_index(1)];
a = a * a - 3.0 * a;
```

In Ada 2022, you can use a similar renaming:

```
declare
  A renames My_Data (To_Index (1));
begin
  A := A ** 2 - 3.0 * A;
end;
```

Here we use a new short form of the rename declaration, but this still looks too heavy, and even worse, it can't be used for discriminant-dependent components.

109.2 References

- ARM 5.2.1 Target Name Symbols⁵⁵⁵
- AI12-0125-3⁵⁵⁶

⁵⁵⁵ <http://www.ada-auth.org/standards/22aarm/html/AA-5-2-1.html>

⁵⁵⁶ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT>

ENUMERATION REPRESENTATION

Note

Enumeration representation attributes are supported by

- GNAT Community Edition 2019
- GCC 9

Enumeration types in Ada are represented as integers at the machine level. But there are actually two mappings from enumeration to integer: a literal position and a representation value.

110.1 Literal positions

Each enumeration literal has a corresponding position in the type declaration. We can easily obtain it from the `Type'Pos` (Enum) attribute.

Listing 2474: main.adb

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3
4 procedure Main is
5 begin
6   Ada.Text_IO.Put ("Pos(False) =");
7   Ada.Integer_Text_IO.Put (Boolean'Pos (False));
8   Ada.Text_IO.New_Line;
9   Ada.Text_IO.Put ("Pos(True) =");
10  Ada.Integer_Text_IO.Put (Boolean'Pos (True));
11 end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Enum_Val.Pos
MD5: de7c39f83f7df231dd648606579996a8

Runtime output

```
Pos(False) =      0
Pos(True)  =      1
```

For the reverse mapping, we use `Type'Val` (Int):

Listing 2475: main.adb

```

1 with Ada.Text_IO;
2
```

(continues on next page)

(continued from previous page)

```

3 procedure Main is
4 begin
5   Ada.Text_IO.Put_Line (Boolean'Val (0)'Image);
6   Ada.Text_IO.Put_Line (Boolean'Val (1)'Image);
7 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Enum_Val.Val
MD5: 43f712d25552970bccc4c0c84089d927

Runtime output

```

FALSE
TRUE

```

110.2 Representation values

The representation value defines the *internal* code, used to store enumeration values in memory or CPU registers. By default, enumeration representation values are the same as the corresponding literal positions, but you can redefine them. Here, we created a copy of **Boolean** type and assigned it a custom representation.

In Ada 2022, we can get an integer value of the representation with **Type'Enum_Rep**(Enum) attribute:

Listing 2476: main.adb

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3
4 procedure Main is
5   type My_Boolean is new Boolean;
6   for My_Boolean use (False => 3, True => 6);
7 begin
8   Ada.Text_IO.Put ("Enum_Rep(False) =");
9   Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (False));
10  Ada.Text_IO.New_Line;
11  Ada.Text_IO.Put ("Enum_Rep(True)  =");
12  Ada.Integer_Text_IO.Put (My_Boolean'Enum_Rep (True));
13 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Enum_Val.Enum_Rep
MD5: 384ad9de7124c8131aa83ab71da58964

Runtime output

```

Enum_Rep(False) =      3
Enum_Rep(True)  =      6

```

And, for the reverse mapping, we can use **Type'Enum_Val** (Int):

Listing 2477: main.adb

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3

```

(continues on next page)

(continued from previous page)

```

4 procedure Main is
5   type My_Boolean is new Boolean;
6   for My_Boolean use (False => 3, True => 6);
7 begin
8   Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (3)'Image);
9   Ada.Text_IO.Put_Line (My_Boolean'Enum_Val (6)'Image);
10
11   Ada.Text_IO.Put ("Pos(False) =");
12   Ada.Integer_Text_IO.Put (My_Boolean'Pos (False));
13   Ada.Text_IO.New_Line;
14   Ada.Text_IO.Put ("Pos(True) =");
15   Ada.Integer_Text_IO.Put (My_Boolean'Pos (True));
16 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Enum_Val.Enum_Val
MD5: 6e06202472d4cf0ea7c68461ac7afcb1

Runtime output

```

FALSE
TRUE
Pos(False) =          0
Pos(True)  =          1

```

Note that the 'Val(X)/'Pos(X) behaviour still is the same.

Custom representations can be useful for integration with a low level protocol or hardware.

110.3 Before Ada 2022

This doesn't initially look like an important feature, but let's see how we'd do the equivalent with Ada 2012 and earlier versions. First, we need an integer type of matching size, then we instantiate `Ada.Unchecked_Conversion`. Next, we call `To_Int/From_Int` to work with representation values. And finally an extra type conversion is needed:

Listing 2478: main.adb

```

1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3 with Ada.Unchecked_Conversion;
4
5 procedure Main is
6
7   type My_Boolean is new Boolean;
8   for My_Boolean use (False => 3, True => 6);
9   type My_Boolean_Int is range 3 .. 6;
10  for My_Boolean_Int'Size use My_Boolean'Size;
11
12  function To_Int is new Ada.Unchecked_Conversion
13    (My_Boolean, My_Boolean_Int);
14
15  function From_Int is new Ada.Unchecked_Conversion
16    (My_Boolean_Int, My_Boolean);
17
18 begin
19   Ada.Text_IO.Put ("To_Int(False) =");
20   Ada.Integer_Text_IO.Put (Integer (To_Int (False)));

```

(continues on next page)

(continued from previous page)

```
21 Ada.Text_IO.New_Line;
22 Ada.Text_IO.Put ("To_Int(True) =");
23 Ada.Integer_Text_IO.Put (Integer (To_Int (True)));
24 Ada.Text_IO.New_Line;
25 Ada.Text_IO.Put ("From_Int (3) =");
26 Ada.Text_IO.Put_Line (From_Int (3)'Image);
27 Ada.Text_IO.New_Line;
28 Ada.Text_IO.Put ("From_Int (6) =");
29 Ada.Text_IO.Put_Line (From_Int (6)'Image);
30 end Main;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Enum_Val.Conv
MD5: 7c7624ed024b26036389f77dbd6cb109

Runtime output

```
To_Int(False) =      3
To_Int(True)  =      6
From_Int (3)  =FALSE

From_Int (6)  =TRUE
```

Even with all that, this solution doesn't work for generic formal type (because T'Size must be a static value)!

We should note that these new attributes may already be familiar to GNAT users because they've been in the GNAT compiler for many years.

110.4 References

- [ARM 13.4 Enumeration Representation Clauses](#)⁵⁵⁷
- [AI12-0237-1](#)⁵⁵⁸

⁵⁵⁷ <http://www.ada-auth.org/standards/22aarm/html/AA-13-4.html>

⁵⁵⁸ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT>

BIG NUMBERS

Note

Big numbers are supported by

- GNAT Community Edition 2020
- GCC 11
- GCC 10 (draft, no user defined literals)

Ada 2022 introduces big integers and big real types.

111.1 Big Integers

The package `Ada.Numerics.Big_Numbers.Big_Integers` contains a type `Big_Integer` and corresponding operations such as comparison (`=`, `<`, `>`, `<=`, `>=`), arithmetic (`+`, `-`, `*`, `/`, `rem`, `mod`, `abs`, `**`), `Min`, `Max` and `Greatest_Common_Divisor`. The type also has `Integer_Literal` and `Put_Image` aspects redefined, so you can use it in a natural manner.

```
Ada.Text_IO.Put_Line (Big_Integer'Image(2 ** 256));
```

```
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

111.2 Tiny RSA implementation

Note

Note that you shouldn't use `Big_Numbers` for cryptography because it's vulnerable to timing side-channels attacks.

We can implement the [RSA algorithm](https://en.wikipedia.org/wiki/RSA_algorithm)⁵⁵⁹ in a few lines of code. The main operation of RSA is $(m^d) \bmod n$. But you can't just write `m ** d`, because these are really big numbers and the result won't fit into memory. However, if you keep intermediate result `mod n` during the m^d calculation, it will work. Let's write this operation as a function:

Listing 2479: `power_mod.ads`

```
1 with Ada.Numerics.Big_Numbers.Big_Integers;  
2 use Ada.Numerics.Big_Numbers.Big_Integers;
```

(continues on next page)

⁵⁵⁹ [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

(continued from previous page)

```

3
4  -- Calculate M ** D mod N
5
6  function Power_Mod (M, D, N : Big_Integer) return Big_Integer;
```

Listing 2480: power_mod.adb

```

1  function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
2
3      function Is_Odd (X : Big_Integer) return Boolean is
4          (X mod 2 /= 0);
5
6      Result : Big_Integer := 1;
7      Exp    : Big_Integer := D;
8      Mult   : Big_Integer := M mod N;
9  begin
10     while Exp /= 0 loop
11         -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
12         if Is_Odd (Exp) then
13             Result := (Result * Mult) mod N;
14         end if;
15
16         Mult := Mult ** 2 mod N;
17         Exp := Exp / 2;
18     end loop;
19
20     return Result;
21 end Power_Mod;
```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Big_Integers
MD5: 8ade78366bf7c98090ae32f9a9830cf9

Let's check this with the example from [Wikipedia](https://en.wikipedia.org/wiki/RSA_(cryptosystem))⁵⁶⁰. In that example, the *public key* is (n = 3233, e = 17) and the message is m = 65. The encrypted message is $m^e \bmod n = 65^{17} \bmod 3233 = 2790 = c$.

```
Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
```

```
2790
```

To decrypt it with the public key (n = 3233, d = 413), we need to calculate $c^d \bmod n = 2790^{413} \bmod 3233$:

```
Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
```

```
65
```

So 65 is the original message m. Easy!

Here is the complete code snippet:

Listing 2481: main.adb

```

1  with Ada.Text_IO;
2  with Ada.Numerics.Big_Numbers.Big_Integers;
3  use  Ada.Numerics.Big_Numbers.Big_Integers;
```

(continues on next page)

⁵⁶⁰ [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

(continued from previous page)

```

4
5 procedure Main is
6
7   -- Calculate M ** D mod N
8
9   function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
10
11     function Is_Odd (X : Big_Integer) return Boolean is
12       (X mod 2 /= 0);
13
14     Result : Big_Integer := 1;
15     Exp    : Big_Integer := D;
16     Mult   : Big_Integer := M mod N;
17   begin
18     while Exp /= 0 loop
19       -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
20       if Is_Odd (Exp) then
21         Result := (Result * Mult) mod N;
22       end if;
23
24       Mult := Mult ** 2 mod N;
25       Exp := Exp / 2;
26     end loop;
27
28     return Result;
29   end Power_Mod;
30
31 begin
32   Ada.Text_IO.Put_Line (Big_Integer'Image (2 ** 256));
33   -- Encrypt:
34   Ada.Text_IO.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
35   -- Decrypt:
36   Ada.Text_IO.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
37 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Big_Numbers_Tiny_RSA
MD5: 298c4a82ceafba9654d41b3a9762927d

Runtime output

```

115792089237316195423570985008687907853269984665640564039457584007913129639936
2790
65

```

111.3 Big Reals

In addition to `Big_Integer`, Ada 2022 provides `Big Reals`⁵⁶¹.

111.4 References

- ARM A.5.6 Big Integers⁵⁶²
- ARM A.5.7 Big Reals⁵⁶³

⁵⁶¹ <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-7.html>

⁵⁶² <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-6.html>

⁵⁶³ <http://www.ada-auth.org/standards/22aarm/html/AA-A-5-7.html>

- AI12-0208-1⁵⁶⁴

⁵⁶⁴ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0208-1.TXT>

INTERFACING C VARIADIC FUNCTIONS

Note

Variadic convention is supported by

- GNAT Community Edition 2020
- GCC 11

In C, [variadic functions](#)⁵⁶⁵ take a variable number of arguments and an ellipsis as the last parameter of the declaration. A typical and well-known example is:

```
int printf (const char* format, ...);
```

Usually, in Ada, we bind such a function with just the parameters we want to use:

```
procedure printf_double
  (format : Interfaces.C.char_array;
   value  : Interfaces.C.double)
  with Import,
    Convention      => C,
    External_Name  => "printf";
```

Then we call it as a normal Ada function:

```
printf_double (Interfaces.C.To_C ("Pi=%f"), Ada.Numerics. $\pi$ );
```

Unfortunately, doing it this way doesn't always work because some [ABI](#)⁵⁶⁶s use different calling conventions for variadic functions. For example, the [AMD64 ABI](#)⁵⁶⁷ specifies:

- %rax — with variable arguments passes information about the number of vector registers used;
- %xmm0—%xmm1 — used to pass and return floating point arguments.

This means, if we write (in C):

```
printf("%d", 5);
```

The compiler will place 0 into %rax, because we don't pass any float argument. But in Ada, if we write:

```
procedure printf_int
  (format : Interfaces.C.char_array;
   value  : Interfaces.C.int)
```

(continues on next page)

⁵⁶⁵ <https://en.cppreference.com/w/c/variadic>

⁵⁶⁶ https://en.wikipedia.org/wiki/Application_binary_interface

⁵⁶⁷ <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

(continued from previous page)

```

with Import,
  Convention    => C,
  External_Name => "printf";

printf_int (Interfaces.C.To_C ("d=%d"), 5);

```

the compiler won't use the %rax register at all. (You can't include any float argument because there's no float parameter in the Ada wrapper function declaration.) As result, you will get a crash, stack corruption, or other undefined behavior.

To fix this, Ada 2022 provides a new family of calling convention names — `C_Variadic_N`:

The convention `C_Variadic_n` is the calling convention for a variadic C function taking n fixed parameters and then a variable number of additional parameters.

Therefore, the correct way to bind the `printf` function is:

```

procedure printf_int
  (format : Interfaces.C.char_array;
   value  : Interfaces.C.int)
with Import,
  Convention    => C_Variadic_1,
  External_Name => "printf";

```

And the following call won't crash on any supported platform:

```
printf_int (Interfaces.C.To_C ("d=%d"), 5);
```

Without this convention, problems caused by this mismatch can be very hard to debug. So, this is a very useful extension to the Ada-to-C interfacing facility.

Here is the complete code snippet:

Listing 2482: main.adb

```

1 with Interfaces.C;
2
3 procedure Main is
4
5   procedure printf_int
6     (format : Interfaces.C.char_array;
7      value  : Interfaces.C.int)
8   with Import,
9     Convention    => C_Variadic_1,
10    External_Name => "printf";
11
12 begin
13   printf_int (Interfaces.C.To_C ("d=%d"), 5);
14 end Main;

```

Code block metadata

Project: Courses.Ada_2022_Whats_New.Variadic_Import
MD5: 94515f55a93f27e4f4ecec31256645d9

112.1 References

- ARM B.3 Interfacing with C and C++⁵⁶⁸

⁵⁶⁸ <http://www.ada-auth.org/standards/22aarm/html/AA-B-3.html>

- AI12-0028-1⁵⁶⁹

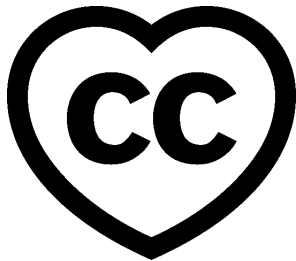
⁵⁶⁹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT>

Part XI

Introduction to the GNAT Toolchain

Copyright © 2019 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)⁵⁷⁰



This course presents an introduction to the GNAT toolchain. The course includes first steps to get started with the toolchain and some details on the project manager (GPRbuild) and the integrated development environment (GNAT Studio).

This document was written by Gustavo A. Hoffmann, with contributions and review from Richard Kenner and Robert Duff.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

⁵⁷⁰ [http://creativecommons.org/licenses/by-sa/4.0](https://creativecommons.org/licenses/by-sa/4.0)

GNAT TOOLCHAIN BASICS

This chapter presents a couple of basic commands from the GNAT toolchain.

113.1 Basic commands

Now that the toolchain is installed, you can start using it. From the command line, you can compile a project using **gprbuild**. For example:

```
gprbuild -P project.gpr
```

You can find the binary built with the command above in the *obj* directory. You can run it in the same way as you would do with any other executable on your platform. For example:

```
obj/main
```

A handy command-line option for **gprbuild** you might want to use is *-p*, which automatically creates directories such as *obj* if they aren't in the directory tree:

```
gprbuild -p -P project.gpr
```

Ada source-code are stored in *.ads* and *.adb* files. To view the content of these files, you can use **GNAT Studio**. To open **GNAT Studio**, double-click on the *.gpr* project file or invoke **GNAT Studio** on the command line:

```
gps -P project.gpr
```

To compile your project using **GNAT Studio**, use the top-level menu to invoke Build → Project → main.adb (or press the keyboard shortcut F4). To run the main program, click on Build → Run → main (or press the keyboard shortcut Shift + F2).

113.2 Compiler warnings

One of the strengths of the GNAT compiler is its ability to generate many useful warnings. Some are displayed by default but others need to be explicitly enabled. In this section, we discuss some of these warnings, their purpose, and how you activate them.

113.2.1 -gnatwa switch and warning suppression

Section author: Robert Duff

We first need to understand the difference between a *warning* and an *error*. Errors are violations of the Ada language rules as specified in the Ada Reference Manual; warnings don't indicate violations of those rules, but instead flag constructs in a program that seem suspicious to the compiler. Warnings are GNAT-specific, so other Ada compilers might not warn about the same things GNAT does or might warn about them in a different way. Warnings

are typically conservative; meaning that some warnings are false alarms. The programmer needs to study the code to determine if each warning is describing a real problem.

Some warnings are produced by default while others are produced only if a switch enables them. Use the `-gnatwa` switch to turn on (almost) all warnings.

Warnings are useless if you don't do anything about them. If you give your team member some code that causes warnings, how are they supposed to know whether they represent real problems? If you don't address each warning, people will soon start ignoring warnings and there'll be lots of things that generate warnings scattered all over your code. To avoid this, you may want to use the `-gnatwae` switch to both turn on (almost) all warnings and to treat warnings as errors. This forces you to get a clean (no warnings or errors) compilation.

However, as we said, some warnings are false alarms. Use `pragma Warnings (Off)` to suppress those warnings. It's best to be as specific as possible and narrow down to a single line of code and a single warning. Then use a comment to explain why the warning is a false alarm if it's not obvious.

Let's look at the following example:

```
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is

  procedure Mumble (X : Integer) is
  begin
    Put_Line ("Mumble processing...");
  end Mumble;

end Warnings_Example;
```

We compile the above code with `-gnatwae`:

```
gnat compile -gnatwae ./src/warnings_example.adb
```

This causes GNAT to complain:

```
warnings_example.adb:5:22: warning: formal parameter "X" is not referenced
```

But the following compiles cleanly:

```
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is

  pragma Warnings (Off, "formal parameter ""X"" is not referenced");
  procedure Mumble (X : Integer) is
  pragma Warnings (On, "formal parameter ""X"" is not referenced");

  -- X is ignored here, because blah blah blah...
  begin
    Put_Line ("Mumble processing...");
  end Mumble;

end Warnings_Example;
```

Here we've suppressed a specific warning message on a specific line.

If you get many warnings of a specific type and it's not feasible to fix all of them, you can suppress that type of message so the good warnings won't get buried beneath a pile of bogus ones. For example, you can use the `-gnatwaeF` switch to silence the warning on the first version of Mumble above: the F suppresses warnings on unreferenced formal parameters. It would be a good idea to use it if you have many of those.

As discussed above, `-gnatwa` activates almost all warnings, but not all. Refer to the [section on warnings](#)⁵⁷¹ of the GNAT User's Guide to get a list of the remaining warnings you could enable in your project. One is `-gnatw.o`, which displays warnings when the compiler detects modified but unreferenced **out** parameters. Consider the following example:

```
package Warnings_Example is

  procedure Process (X : in out Integer;
                    B : out Boolean);

end Warnings_Example;
```

```
package body Warnings_Example is

  procedure Process (X : in out Integer;
                    B : out Boolean) is
  begin
    if X = Integer'First or else X = Integer'Last then
      B := False;
    else
      X := X + 1;
      B := True;
    end if;
  end Process;

end Warnings_Example;
```

```
with Ada.Text_IO; use Ada.Text_IO;

with Warnings_Example; use Warnings_Example;

procedure Main is
  X : Integer := 0;
  Success : Boolean;
begin
  Process (X, Success);
  Put_Line (Integer'Image (X));
end Main;
```

If we build the main application using the `-gnatw.o` switch, the compiler warns us that we didn't reference the `Success` variable, which was modified in the call to `Process`:

```
main.adb:8:16: warning: "Success" modified by call, but value might not be
↳referenced
```

In this case, this actually points us to a bug in our program, since `X` only contains a valid value if `Success` is **True**. The corrected code for `Main` is:

```
-- ...
begin
  Process (X, Success);

  if Success then
    Put_Line (Integer'Image (X));
  else
    Put_Line ("Couldn't process variable X.");
  end if;
end Main;
```

⁵⁷¹ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#warning-message-control

We suggest turning on as many warnings as makes sense for your project. Then, when you see a warning message, look at the code and decide if it's real. If it is, fix the code. If it's a false alarm, suppress the warning. In either case, we strongly recommend you make the warning disappear before you check your code into your configuration management system.

113.2.2 Style checking

GNAT provides many options to configure style checking of your code. The main compiler switch for this is `-gnatyy`, which sets almost all standard style check options. As indicated by the [section on style checking](#)⁵⁷² of the GNAT User's Guide, using this switch "is equivalent to `-gnaty3aAbcefhiklmprst`, that is all checking options enabled with the exception of `-gnatyB`, `-gnatyD`, `-gnatyI`, `-gnatyLnnn`, `-gnatyO`, `-gnaty0`, `-gnatyS`, `-gnatyU`, and `-gnatYx`."

You may find that selecting the appropriate coding style is useful to detect issues at early stages. For example, the `-gnaty0` switch checks that overriding subprograms are explicitly marked as such. Using this switch can avoid surprises when you didn't intentionally want to override an operation for some data type. We recommend studying the list of coding style switches and selecting the ones that seem relevant for your project. When in doubt, you can start by using all of them — using `-gnatyy` and `-gnatyBdIL4o0Sux`, for example — and deactivating the ones that cause too much *noise* during compilation.

⁵⁷² https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#style-checking

GPRBUILD

This chapter presents a brief overview of **GPRbuild**, the project manager of the GNAT toolchain. It can be used to manage complex builds. In terms of functionality, it's similar to **make** and **cmake**, just to name two examples.

For a detailed presentation of the tool, please refer to the [GPRbuild User's Guide](#)⁵⁷³.

114.1 Basic commands

As mentioned in the previous chapter, you can build a project using **gprbuild** from the command line:

```
gprbuild -P project.gpr
```

In order to clean the project, you can use **gprclean**:

```
gprclean -P project.gpr
```

114.2 Project files

You can create project files using **GNAT Studio**, which presents many options on its graphical interface. However, you can also edit project files manually as a normal text file in an editor, since its syntax is human readable. In fact, project files use a syntax similar to the one from the Ada language. Let's look at the basic structure of project files and how to customize them.

114.2.1 Basic structure

The main element of a project file is a project declaration, which contains definitions for the current project. A project file may also include other project files in order to compose a complex build. One of the simplest form of a project file is the following:

```
project Default is
  for Main use ("main");
  for Source_Dirs use ("src");
end Default;
```

In this example, we declare a project named `Default`. The `for Main use` expression indicates that the `main.adb` file is used as the entry point (main source-code file) of the project. The main file doesn't necessary need to be called `main.adb`; we could use any source-code implementing a main application, or even have a list of multiple main files.

⁵⁷³ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

The `for Source_Dirs` use expression indicates that the `src` directory contains the source-file for the application (including the main file).

114.2.2 Customization

GPRbuild support scenario variables, which allow you to control the way binaries are built. For example, you may want to distinguish between debug and optimized versions of your binary. In principle, you could pass command-line options to **gprbuild** that turn debugging on and off, for example. However, defining this information in the project file is usually easier to handle and to maintain. Let's define a scenario variable called `ver` in our project:

```
project Default is
    Ver := external ("ver", "debug");

    for Main use ("main");
    for Source_Dirs use ("src");

end Default;
```

In this example, we're specifying that the scenario variable `Ver` is initialized with the external variable `ver`. Its default value is set to `debug`.

We can now set this variable in the call to **gprbuild**:

```
gprbuild -P project.gpr -Xver=debug
```

Alternatively, we can simply specify an environment variable. For example, on Unix systems, we can say:

```
export ver=debug
# Value from environment variable "ver" used in the following call:
gprbuild -P project.gpr
```

In the project file, we can use the scenario variable to customize the build:

```
project Default is
    Ver := external ("ver", "debug");

    for Main use ("main.adb");
    for Source_Dirs use ("src");

    -- Using "ver" variable for obj directory
    for Object_Dir use "obj/" & Ver;

    package Compiler is
        case Ver is
            when "debug" =>
                for Switches ("Ada") use ("-g");
            when "opt" =>
                for Switches ("Ada") use ("-O2");
            when others =>
                null;
        end case;
    end Compiler;

end Default;
```

We're now using `Ver` in the `for Object_Dir` clause to specify a subdirectory of the `obj` directory that contains the object files. Also, we're using `Ver` to select compiler options in

the Compiler package declaration.

We could also specify all available options in the project file by creating a typed variable. For example:

```
project Default is

  type Ver_Option is ("debug", "opt");
  Ver : Ver_Option := external ("ver", "debug");

  for Source_Dirs use ("src");
  for Main use ("main.adb");

  -- Using "ver" variable for obj directory
  for Object_Dir use "obj/" & Ver;

  package Compiler is
    case Ver is
      when "debug" =>
        for Switches ("Ada") use ("-g");
      when "opt" =>
        for Switches ("Ada") use ("-O2");
      when others =>
        null;
    end case;
  end Compiler;

end Default;
```

The advantage of this approach is that **gprbuild** can now check whether the value that you provide for the `ver` variable is available on the list of possible values and give you an error if you're entering a wrong value.

114.3 Project dependencies

GPRbuild supports project dependencies. This allows you to reuse information from existing projects. Specifically, the keyword **with** allows you to include another project within the current project.

114.3.1 Simple dependency

Let's look at a very simple example. We have a package called `Test_Pkg` associated with the project file `test_pkg.gpr`, which contains:

```
project Test_Pkg is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
end Test_Pkg;
```

This is the code for the `Test_Pkg` package:

```
package Test_Pkg is

  type T is record
    X : Integer;
    Y : Integer;
  end record;

  function Init return T;
```

(continues on next page)

(continued from previous page)

```
end Test_Pkg;
```

```
package body Test_Pkg is

  function Init return T is
  begin
    return V : T do
      V.X := 0;
      V.Y := 0;
    end return;
  end Init;

end Test_Pkg;
```

For this example, we use a directory `test_pkg` containing the project file and a subdirectory `test_pkg/src` containing the source files. The directory structure looks like this:

```
| - test_pkg
|   | test_pkg.gpr
|   | - src
|   |   | test_pkg.adb
|   |   | test_pkg.ads
```

Suppose we want to use the `Test_Pkg` package in a new application. Instead of directly including the source files of `Test_Pkg` in the project file of our application (either directly or indirectly), we can instead reference the existing project file for the package by using `with "test_pkg.gpr"`. This is the resulting project file:

```
with "../test_pkg/test_pkg.gpr";

project Default is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Main use ("main.adb");
end Default;
```

And this is the code for the main application:

```
with Test_Pkg; use Test_Pkg;

procedure Main is
  A : T;
begin
  A := Init;
end Main;
```

When we build the main project file (`default.gpr`), we're automatically building all dependent projects. More specifically, the project file for the main application automatically includes the information from the dependent projects such as `test_pkg.gpr`. Using a `with` in the main project file is all we have to do for that to happen.

114.3.2 Dependencies to dynamic libraries

We can structure project files to make use of dynamic (shared) libraries using a very similar approach. It's straightforward to convert the project above so that `Test_Pkg` is now compiled into a dynamic library and linked to our main application. All we need to do is to make a few additions to the project file for the `Test_Pkg` package:

```

library project Test_Pkg is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Library_Name use "test_pkg";
  for Library_Dir use "lib";
  for Library_Kind use "Dynamic";
end Test_Pkg;

```

This is what we had to do:

- We changed the project to `library project`.
- We added the specification for `Library_Name`, `Library_Dir` and `Library_Kind`.

We don't need to change the project file for the main application because **GPRbuild** automatically detects the dependency information (e.g., the path to the dynamic library) from the project file for the `Test_Pkg` package. With these small changes, we're able to compile the `Test_Pkg` package to a dynamic library and link it with our main application.

114.4 Configuration pragma files

Configuration pragma files contain a set of pragmas that modify the compilation of source files according to external requirements. For example, you may use pragmas to either relax or strengthen requirements depending on your environment.

In **GPRbuild**, we can use `Local_Configuration_Pragmas` (in the `Compiler` package) to indicate the configuration pragmas file we want **GPRbuild** to use with the source files in our project.

The file `gnat.adc` shown here is an example of a configuration pragma file:

```
pragma Suppress (Overflow_Check);
```

We can use this in our project by declaring a `Compiler` package. Here's the complete project file:

```

project Default is

  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Main use ("main.adb");

  package Compiler is
    for Local_Configuration_Pragmas use "gnat.adc";
  end Compiler;

end Default;

```

Each pragma contained in `gnat.adc` is used in the compilation of each file, as if that pragma was placed at the beginning of each file.

114.5 Configuration packages

You can control the compilation of your source code by creating variants for various cases and selecting the appropriate variant in the compilation package in the project file. One example where this is useful is conditional compilation using Boolean constants, shown in the code below:

```
with Ada.Text_IO; use Ada.Text_IO;

with Config;

procedure Main is
begin
    if Config.Debug then
        Put_Line ("Debug version");
    else
        Put_Line ("Release version");
    end if;
end Main;
```

In this example, we declared the Boolean constant in the Config package. By having multiple versions of that package, we can create different behavior for each usage. For this simple example, there are only two possible cases: either Debug is **True** or **False**. However, we can apply this strategy to create more complex cases.

In our next example, we store the packages in the subdirectories debug and release of the source code directory. Here's the content of the src/debug/config.ads file:

```
package Config is
    Debug : constant Boolean := True;
end Config;
```

Here's the src/release/config.ads file:

```
package Config is
    Debug : constant Boolean := False;
end Config;
```

In this case, **GPRbuild** selects the appropriate directory to look for the config.ads file according to information we provide for the compilation process. We do this by using a scenario type called Mode_Type in our project file:

```
gprbuild -P default.gpr -Xmode=release
```

```
project Default is
    type Mode_Type is ("debug", "release");

    Mode : Mode_Type := external ("mode", "debug");

    for Source_Dirs use ("src", "src/" & Mode);
    for Object_Dir use "obj";
    for Main use ("main.adb");

end Default;
```

We declare the scenario variable Mode and use it in the Source_Dirs declaration to add the desired path to the subdirectory containing the config.ads file. The expression "src/" & Mode concatenates the user-specified mode to select the appropriate subdirectory. For more complex cases, we could use either a tree of subdirectories or multiple scenario variables for each aspect that we need to configure.

GNAT STUDIO

This chapter presents an introduction to the GNAT Studio, which provides an IDE to develop applications in Ada. For a detailed overview, please refer to the [GNAT Studio tutorial](https://docs.adacore.com/live/wave/gps/html/gps_tutorial/index.html)⁵⁷⁴. Also, you can refer to the [GNAT Studio product page](https://www.adacore.com/gnatpro/toolsuite/gps)⁵⁷⁵ for some introductory videos.

In this chapter, all indications using "→" refer to options from the GNAT Studio menu that you can click in order to execute commands.

115.1 Start-up

The first step is to start-up the GNAT Studio. The actual step depends on your platform.

115.1.1 Windows

- You may find an icon (shortcut to **GNAT Studio**) on your desktop.
- Otherwise, start **GNAT Studio** by typing `gnatstudio` on the command prompt.

115.1.2 Linux

- Start **GNAT Studio** by typing `gnatstudio` on a shell.

115.2 Creating projects

After starting-up **GNAT Studio**, you can create a project. These are the steps:

- Click on **Create new project** in the welcome window
 - Alternatively, if the *wizard* (which let's you customize new projects) isn't already opened, click on **File → New Project...** to open it.
 - After clicking on **Create new project**, you should see a window with this title: **Create Project from Template**.
- Select one of the options from the list and click on **Next**.
 - The simplest one is **Basic > Simple Ada Project**, which creates a project containing a main application.
- Select the project location and basic settings, and click on **Apply**.
 - If you selected "Simple Ada Project" in the previous step, you may now select the name of the project and of the main file.
 - Note that you can select any name for the main file.

⁵⁷⁴ https://docs.adacore.com/live/wave/gps/html/gps_tutorial/index.html

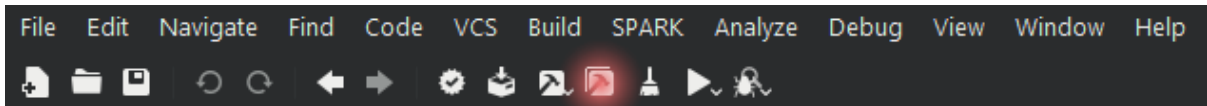
⁵⁷⁵ <https://www.adacore.com/gnatpro/toolsuite/gps>

You should now have a working project file.

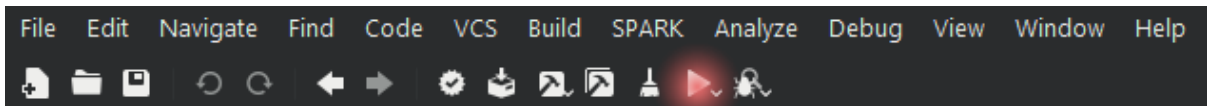
115.3 Building

As soon as you've created a project file, you can use it to build an application. These are the required steps:

- Click on Build → Project → Build All
 - You can also click on this icon:



- Alternatively, you can click on Build → Project → Build & Run → <name of your main application>
 - You can also click on this icon:



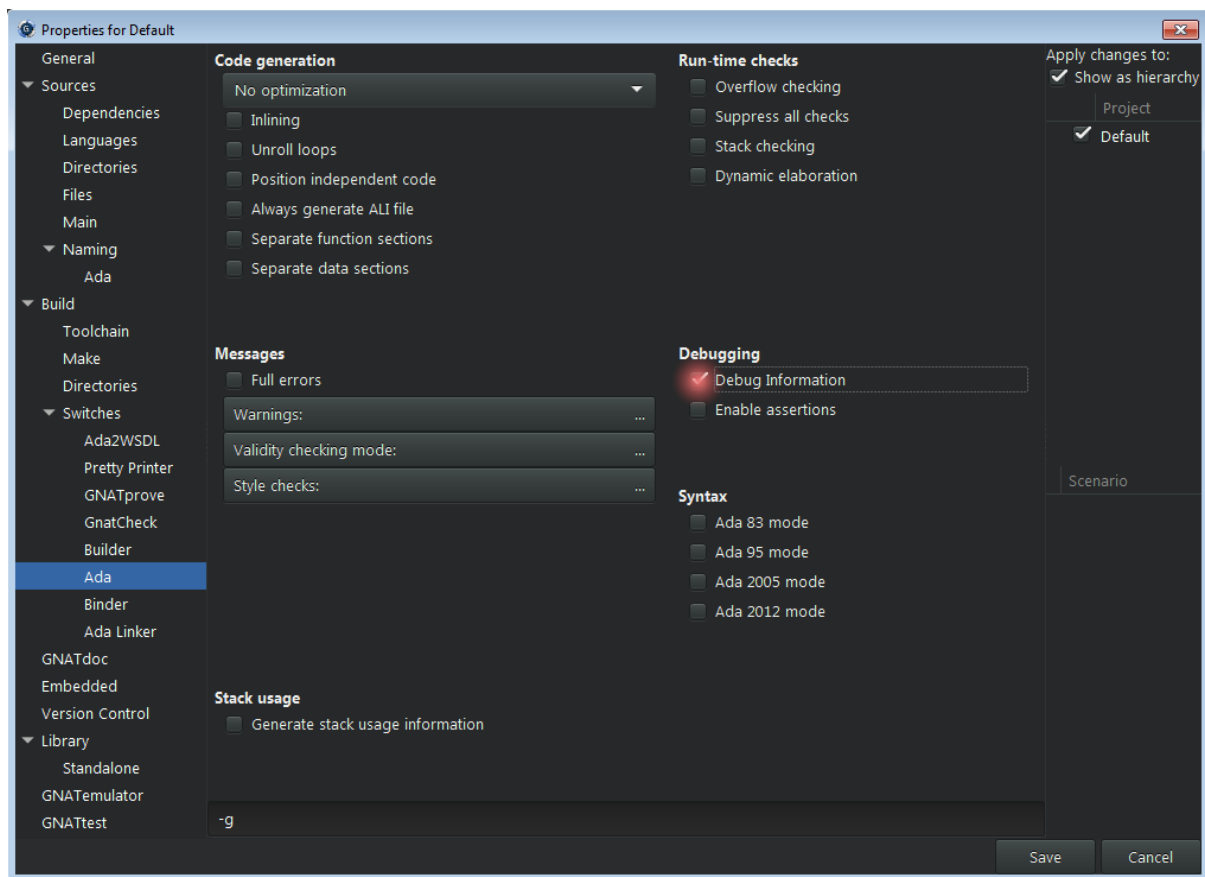
- You can also use the keyboard for building and running the main application:
 - Press F4 to open a window that allows you to build the main application and click on Execute.
 - Then, press Shift + F2 to open a window that allows you to run the application, and click on Execute.

115.4 Debugging

115.4.1 Debug information

Before you can debug a project, you need to make sure that debugging symbols have been included in the binary build. You can do this by manually adding a debug version into your project, as described in the previous chapter (see [GPRbuild](#) (page 2133)).

Alternatively, you can change the project properties directly in **GNAT Studio**. In order to do that, click on Edit → Project Properties..., which opens the following window:



Click on Build → Switches → Ada on this window, and make sure that the Debug Information option is selected.

115.4.2 Improving main application

If you selected "Simple Ada Project" while creating your project in the beginning, you probably still have a very simple main application that doesn't do anything useful. Therefore, in order to make the debugging activity more interesting, please enter some statements to your application. For example:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
  Put_Line ("Hello World!");
  Put_Line ("Hello again!");
end Main;
```

115.4.3 Debugging the application

You can now build and debug the application by clicking on Build → Project → Build & Debug → <name of your main application>.

You can then click on Debug → Run... to open a window that allows you to start the application. Alternatively, you can press Shift + F9. As soon as the application has started, you can press F5 to step through the application or press F6 to execute until the next line. Both commands are available in the menu by clicking on Debug → Step or Debug → Next.

When you've finished debugging your application, you need to terminate the debugger. To do this, you can click on Debug → Terminate.

115.5 Formal verification

In order to see how SPARK can detect issues, let's create a simple application that accumulates values in a variable A:

```

procedure Main
  with SPARK_Mode is

    procedure Acc (A : in out Natural;
                  V :      Natural) is
    begin
      A := A + V;
    end Acc;

    A : Natural := 0;
begin
  Acc (A, Natural'Last);
  Acc (A, 1);
end Main;

```

You can now click on SPARK → Prove All, which opens a window with various options. For example, on this window, you can select the proof level — varying between 0 and 4 — on the Proof level list. Next, click on Execute. After the prover has completed its analysis, you'll see a list of issues found in the source code of your application.

For the example above, the prover complains about an overflow check that might fail. This is due to the fact that, in the Acc procedure, we're not dealing with the possibility that the result of the addition might be out of range. In order to fix this, we could define a new saturating addition Sat_Add that makes use of a custom type T with an extended range. For example:

```

procedure Main
  with SPARK_Mode is

    function Sat_Add (A : Natural;
                    V : Natural) return Natural
    is
      type T is range Natural'First .. Natural'Last * 2;

      A2      : T      := T (A);
      V2      : constant T := T (V);
      A_Last  : constant T := T (Natural'Last);
    begin
      A2 := A2 + V2;

      -- Saturate result if needed
      if A2 > A_Last then
        A2 := A_Last;
      end if;

      return Natural (A2);
    end Sat_Add;

    procedure Acc (A : in out Natural;
                  V :      Natural) is
    begin
      A := Sat_Add (A, V);
    end Acc;

    A : Natural := 0;
begin
  Acc (A, Natural'Last);

```

(continues on next page)

(continued from previous page)

```
    Acc (A, 1);  
end Main;
```

Now, when running the prover again with the modified code, no issues are found.

GNAT TOOLS

In chapter we present a brief overview of some of the tools included in the GNAT toolchain. For further details on how to use these tools, please refer to the [GNAT User's Guide](https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn.html)⁵⁷⁶.

116.1 gnatchop

gnatchop renames files so they match the file structure and naming convention expected by the rest of the GNAT toolchain. The GNAT compiler expects specifications to be stored in `.ads` files and bodies (implementations) to be stored in `.adb` files. It also expects file names to correspond to the content of each file. For example, it expects the specification of a package `Pkg.Child` to be stored in a file named `pkg-child.ads`.

However, we may not want to use that convention for our project. For example, we may have multiple Ada packages contained in a single file. Consider a file `example.adb` containing the following:

```
with Ada.Text_IO; use Ada.Text_IO;

package P is
  procedure Test;
end P;

package body P is
  procedure Test is
  begin
    Put_Line("Test passed.");
  end Test;
end P;

with P; use P;

procedure P_Main is
begin
  P.Test;
end P_Main;
```

To compile this code, we first pass the file containing our source code to **gnatchop** before we call **gprbuild**:

```
gnatchop example.adb
gprbuild p_main
```

This generates source files for our project, extracted from `example.adb`, that conform to the default naming convention and then builds the executable binary `p_main` from those files. In this example **gnatchop** created the files `p.ads`, `p.adb`, and `p_main.adb` using the package names in `example.adb`.

⁵⁷⁶ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn.html

When we use this mechanism, any warnings or errors the compiler displays refers to the files generated by **gnatchop**. We can, however, instruct **gnatchop** to instrument the generated files so the compiler refers to the original file (`example.ada` in our case) when displaying messages. We do this by using the `-r` switch:

```
gnatchop -r example.ada
gprbuild p_main
```

If, for example, we had an unused variable in `example.ada`, the compiler warning would now refer to the line in the original file, not in one of the generated ones.

For documentation of other switches available for **gnatchop**, please refer to the [gnatchop chapter⁵⁷⁷](#) of the GNAT User's Guide.

116.2 gnatprep

We may want to use conditional compilation in some situations. For example, we might need a customized implementation of a package for a specific platform or need to select a specific version of an algorithm depending on the requirements of the target environment. A traditional way to do this uses a source-code preprocessor. However, in many cases where conditional compilation is needed, we can instead use the syntax of the Ada language or the functionality provided by **GPRbuild** to avoid using a preprocessor in those cases. The [conditional compilation section⁵⁷⁸](#) of the GNAT User's Guide discusses how to do this in detail.

Nevertheless, using a preprocessor is often the most straightforward option in complex cases. When we encounter such a case, we can use **gnatprep**, which provides a syntax that reminds us of the C and C++ preprocessor. However, unlike in C and C++, this syntax is not part of the Ada standard and can only be used with **gnatprep**. Also, you'll notice some differences in the syntax from that preprocessor, such as shown in the example below:

```
#if VERSION'Defined and then (VERSION >= 4) then
  -- Implementation for version 4.0 and above...
#else
  -- Standard implementation for older versions...
#end if;
```

Of course, in this simple case, we could have used the Ada language directly and avoided the preprocessor entirely:

```
package Config is
  Version : constant Integer := 4;
end Config;

with Config;
procedure Do_Something is
begin
  if Config.Version >= 4 then
    null;
    -- Implementation for version 4.0 and above...
  else
    null;
    -- Standard implementation for older versions...
  end if;
end Do_Something;
```

⁵⁷⁷ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#renaming-files-with-gnatchop

⁵⁷⁸ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#conditional-compilation

But for the sake of illustrating the use of **gnatprep**, let's use that tool in this simple case. This is the complete procedure, which we place in file `do_something.org.adb`:

```
procedure Do_Something is
begin
  #if VERSION'Defined and then (VERSION >= 4) then
    -- Implementation for version 4.0 and above...
    null;
  #else
    -- Standard implementation for older versions...
    null;
  #end if;
end Do_Something;
```

To preprocess this file and build the application, we call **gnatprep** followed by **GPRbuild**:

```
gnatprep do_something.org.adb do_something.adb
gprbuild do_something
```

If we look at the resulting file after preprocessing, we see that the `#else` implementation was selected by **gnatprep**. To cause it to select the newer "version" of the code, we include the symbol and its value in our call to **gnatprep**, just like we'd do for C/C++:

```
gnatprep -DVERSION=5 do_something.org.adb do_something.adb
```

However, a cleaner approach is to create a symbol definition file containing all symbols we use in our implementation. Let's create the file and name it `prep.def`:

```
VERSION := 5
```

Now we just need to pass it to **gnatprep**:

```
gnatprep do_something.org.adb do_something.adb prep.def
gprbuild do_something
```

When we use **gnatprep** in that way, the line numbers of the output file differ from those of the input file. To preserve line numbers, we can use one of these command-line switches:

- `-b`: replace stripped-out code by blank lines
- `-c`: comment-out the stripped-out code

For example:

```
gnatprep -b do_something.org.adb do_something.adb prep.def
gnatprep -c do_something.org.adb do_something.adb prep.def
```

When we use one of these options, **gnatprep** ensures that the output file `do_something.adb` has the same line numbering as the original file (`do_something.org.adb`).

The [gnatprep chapter⁵⁷⁹](https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#preprocessing-with-gnatprep) of the GNAT User's Guide contains further details about this tool, such as how to integrate **gnatprep** with project files for **GPRbuild** and how to replace symbols without using preprocessing directives (using the `$symbol` syntax).

116.3 gnatmem

Memory allocation errors involving mismatches between allocations and deallocations are a common source of memory leaks. To test an application for memory allocation issues, we can use **gnatmem**. This tool monitors all memory allocations in our application. We use

⁵⁷⁹ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#preprocessing-with-gnatprep

this tool by linking our application to a special version of the memory allocation library (libgmem.a).

Let's consider this simple example:

```
procedure Simple_Mem is
  I_Ptr : access Integer := new Integer;
begin
  null;
end Simple_Mem;
```

To generate a memory report for this code, we need to:

- Build the application, linking it to libgmem.a;
- Run the application, which generates an output file (gmem.out);
- Run **gnatmem** to generate a report from gmem.out.

For our example above, we do the following:

```
# Build application using gmem
gnatmake -g simple_mem.adb -largS -lgmem

# Run the application and generate gmem.out
./simple_mem

# Call gnatmem to display the memory report based on gmem.out
gnatmem simple_mem
```

For this example, **gnatmem** produces the following output:

```
Global information
-----
Total number of allocations      : 1
Total number of deallocations   : 0
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark                 : 4 Bytes

Allocation Root # 1
-----
Number of non freed allocations  : 1
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark                 : 4 Bytes
Backtrace                       :
    simple_mem.adb:2 simple_mem
```

This shows all the memory we allocated and tells us that we didn't deallocate any of it.

Please refer to the [chapter on gnatmem](#)⁵⁸⁰ of the GNAT User's Guide for a more detailed discussion of **gnatmem**.

116.4 gnatdoc

Use **GNATdoc** to generate HTML documentation for your project. It scans the source files in the project and extracts information from package, subprogram, and type declarations.

The simplest way to use it is to provide the name of the project or to invoke **GNATdoc** from a directory containing a project file:

⁵⁸⁰ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_and_program_execution.html#the-gnatmem-tool

```
gnatdoc -P some_directory/default.gpr
```

Alternatively, when the :file:`default.gpr` file is in the same directory

```
gnatdoc
```

Just using this command is sufficient if your goal is to generate a list of the packages and a list of subprograms in each. However, to create more meaningful documentation, you can annotate your source code to add a description of each subprogram, parameter, and field. For example:

```
package P is
-- Collection of auxiliary subprograms

function Add_One
(V : Integer
-- Coefficient to be incremented
) return Integer;
-- @return Coefficient incremented by one

end P;
```

```
package body P is

function Add_One (V : Integer) return Integer is
begin
return V + 1;
end Add_One;

end P;
```

```
with P; use P;

procedure Main is

I : Integer;

begin
I := Add_One (0);
end Main;
```

When we run this example, **GNATdoc** will extract the documentation from the specification of package P and add the description of each element, which we provided as a comment in the line below the actual declaration. It will also extract the package description, which we wrote as a comment in the line right after **package P is**. Finally, it will extract the documentation of function Add_One (both the description of the V parameter and the return value).

In addition to the approach we've just seen, **GNATdoc** also supports the tagged format that's commonly found in tools such as Javadoc and uses the @ syntax. We could rewrite the documentation for package P as follows:

```
package P is
-- @summary Collection of auxiliary subprograms

function Add_One
(V : Integer
) return Integer;
-- @param V Coefficient to be incremented
-- @return Coefficient incremented by one
```

(continues on next page)

(continued from previous page)

```
end P;
```

You can control what parts of the source-code **GNATdoc** parses to extract the documentation. For example, you can specify the `-b` switch to request that the package body be parsed for additional documentation and you can use the `-p` switch to request **GNATdoc** to parse the private part of package specifications. For a complete list of switches, please refer to the [GNATdoc User's Guide](http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/index.html)⁵⁸¹.

116.5 gnatpp

The term 'pretty-printing' refers to the process of formatting source code according to a pre-defined convention. **gnatpp** is used for the pretty-printing of Ada source-code files.

Let's look at this example, which contains very messy formatting:

```
PrOcEDuRE Main
    IS
    FUNCTION
        Init_2
        RETURN
        inteGER
        is
        (2);
        I : INTeGer;

    BeGiN
        I := Init_2;
    ENd;
```

We can request **gnatpp** to clean up this file by using the command:

```
gnatpp main.adb
```

gnatpp reformats the file in place. After this command, `main.adb` looks like this:

```
procedure Main is
  function Init_2 return Integer is (2);
  I : Integer;
begin
  I := Init_2;
end Main;
```

We can also process all source code files from a project at once by specifying a project file. For example:

```
gnatpp -P default.gpr
```

gnatpp has an extensive list of options, which allow for specifying the formatting of many

⁵⁸¹ http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/index.html

aspects of the source and implementing many coding styles. These are extensively discussed in the [section on gnatpp](#)⁵⁸² of the GNAT User's Guide.

116.6 gnatstub

Suppose you've created a complex specification of an Ada package. You can create the corresponding package body by copying and adapting the content of the package specification. But you can also have **gnatstub** do much of that job for you. For example, let's consider the following package specification:

```
package Aux is

  function Add_One (V : Integer) return Integer;

  procedure Reset (V : in out Integer);

end Aux;
```

We call **gnatstub**, passing the file containing the package specification:

```
gnatstub aux.ads
```

This generates the file `aux.adb` with the following contents:

```
pragma Ada_2012;
package body Aux is

  -----
  -- Add_One --
  -----

  function Add_One (V : Integer) return Integer is
  begin
    -- Generated stub: replace with real body!
    pragma Compile_Time_Warning (Standard.True, "Add_One unimplemented");
    return raise Program_Error with "Unimplemented function Add_One";
  end Add_One;

  -----
  -- Reset --
  -----

  procedure Reset (V : in out Integer) is
  begin
    -- Generated stub: replace with real body!
    pragma Compile_Time_Warning (Standard.True, "Reset unimplemented");
    raise Program_Error with "Unimplemented procedure Reset";
  end Reset;

end Aux;
```

As we can see in this example, not only has **gnatstub** created a package body from all the elements in the package specification, but it also created:

- Headers for each subprogram (as comments);
- Pragmas and exceptions that prevent us from using the unimplemented subprograms in our application.

⁵⁸² https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-pretty-printer-gnatpp

This is a good starting point for the implementation of the body. Please refer to the [section on gnatstub](#)⁵⁸³ of the GNAT User's Guide for a detailed discussion of **gnatstub** and its options.

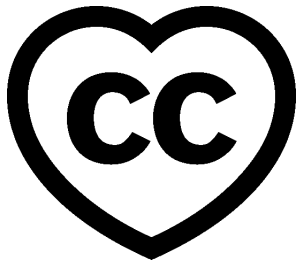
⁵⁸³ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-body-stub-generator-gnatstub

Part XII

Introduction to Ada: Laboratories

Copyright © 2019 – 2024, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](https://creativecommons.org/licenses/by-sa/4.0)⁵⁸⁴



These labs contain exercises for the [Introduction to Ada](#) (page 5) course.

This document was written by Gustavo A. Hoffmann and reviewed by Michael Frank.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

⁵⁸⁴ <http://creativecommons.org/licenses/by-sa/4.0>

IMPERATIVE LANGUAGE

For the exercises below (except for the first one), don't worry about the details of the Main procedure. You should just focus on implementing the application in the subprogram specified by the exercise.

117.1 Hello World

Goal: create a "Hello World!" application.

Steps:

1. Complete the Main procedure.

Requirements:

1. The application must display the message "Hello World!".

Remarks:

1. The part that you have to modify is indicated by the *-- Implement the application here!* comment in the source code.

Listing 2483: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     -- Implement the application here!
6     null;
7 end Main;
```

117.2 Greetings

Goal: create an application that greets a person.

Steps:

1. Complete the Greet procedure.

Requirements:

1. Given an input string <name>, procedure Greet must display the message "Hello <name>!".
 1. For example, if the name is "John", it displays the message "Hello John!".

Remarks:

1. You can use the concatenation operator (&).

2. The part that you have to modify is indicated by the *-- Implement the application here!* comment in the source code.

Listing 2484: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Greet (Name : String) is
7     begin
8         -- Implement the application here!
9         null;
10    end Greet;
11
12 begin
13     if Argument_Count < 1 then
14         Put_Line ("ERROR: missing arguments! Exiting...");
15         return;
16     elsif Argument_Count > 1 then
17         Put_Line ("Ignoring additional arguments...");
18     end if;
19
20     Greet (Argument (1));
21 end Main;
```

117.3 Positive Or Negative

Goal: create an application that classifies integer numbers.

Steps:

1. Complete the Classify_Number procedure.

Requirements:

1. Given an integer number X, procedure Classify_Number must classify X as positive, negative or zero and display the result:
 1. If $X > 0$, it displays Positive.
 2. If $X < 0$, it displays Negative.
 3. If $X = 0$, it displays Zero.

Listing 2485: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 2486: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     -- Implement the application here!
6     null;
7 end Classify_Number;
```

Listing 2487: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Classify_Number;
5
6 procedure Main is
7   A : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
14   end if;
15
16   A := Integer'Value (Argument (1));
17
18   Classify_Number (A);
19 end Main;

```

117.4 Numbers

Goal: create an application that displays numbers in a specific order.

Steps:

1. Complete the Display_Numbers procedure.

Requirements:

1. Given two integer numbers, Display_Numbers displays all numbers in the range starting with the smallest number.

Listing 2488: display_numbers.ads

```

1 procedure Display_Numbers (A, B : Integer);

```

Listing 2489: display_numbers.adb

```

1 procedure Display_Numbers (A, B : Integer) is
2 begin
3   -- Implement the application here!
4   null;
5 end Display_Numbers;

```

Listing 2490: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Numbers;
5
6 procedure Main is
7   A, B : Integer;
8 begin
9   if Argument_Count < 2 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;

```

(continues on next page)

(continued from previous page)

```
12  elsif Argument_Count > 2 then
13      Put_Line ("Ignoring additional arguments...");
14  end if;
15
16  A := Integer'Value (Argument (1));
17  B := Integer'Value (Argument (2));
18
19  Display_Numbers (A, B);
20 end Main;
```

SUBPROGRAMS

118.1 Subtract procedure

Goal: write a procedure that subtracts two numbers.

Steps:

1. Complete the procedure Subtract.

Requirements:

1. Subtract performs the operation $A - B$.

Listing 2491: subtract.ads

```
1  -- Write the correct parameters for the procedure below.
2  procedure Subtract;
```

Listing 2492: subtract.adb

```
1  procedure Subtract is
2  begin
3      -- Implement the procedure here.
4      null;
5  end Subtract;
```

Listing 2493: main.adb

```
1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          Result : Integer;
15      begin
16          case TC is
17              when Sub_10_1_Chk =>
18                  Subtract (10, 1, Result);
19                  Put_Line ("Result: " & Integer'Image (Result));
20              when Sub_10_100_Chk =>
21                  Subtract (10, 100, Result);
```

(continues on next page)

(continued from previous page)

```

22     Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Subtract (0, 5, Result);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Subtract (0, -5, Result);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

118.2 Subtract function

Goal: write a function that subtracts two numbers.

Steps:

1. Rewrite the Subtract procedure from the previous exercise as a function.

Requirements:

1. Subtract performs the operation $A - B$ and returns the result.

Listing 2494: subtract.ads

```

1  -- Write the correct signature for the function below.
2  -- Don't forget to replace the keyword "procedure" by "function."
3  procedure Subtract;

```

Listing 2495: subtract.adb

```

1  procedure Subtract is
2  begin
3      -- Implement the function here!
4      null;
5  end Subtract;

```

Listing 2496: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);

```

(continues on next page)

(continued from previous page)

```

12
13 procedure Check (TC : Test_Case_Index) is
14     Result : Integer;
15 begin
16     case TC is
17     when Sub_10_1_Chk =>
18         Result := Subtract (10, 1);
19         Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21         Result := Subtract (10, 100);
22         Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Result := Subtract (0, 5);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Result := Subtract (0, -5);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

118.3 Equality function

Goal: write a function that compares two values and returns a flag.

Steps:

1. Complete the Is_Equal subprogram.

Requirements:

1. Is_Equal returns a flag as a **Boolean** value.
2. The flag must indicate whether the values are equal (flag is **True**) or not (flag is **False**).

Listing 2497: is_equal.ads

```

1  -- Write the correct signature for the function below.
2  -- Don't forget to replace the keyword "procedure" by "function."
3  procedure Is_Equal;

```

Listing 2498: is_equal.adb

```

1  procedure Is_Equal is
2  begin
3      -- Implement the function here!
4      null;
5  end Is_Equal;

```

Listing 2499: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Is_Equal;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Equal_Chk,
9           Inequal_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Display_Equal (A, B : Integer;
14                                 Equal : Boolean) is
15             begin
16                 Put (Integer'Image (A));
17                 if Equal then
18                     Put (" is equal to ");
19                 else
20                     Put (" isn't equal to ");
21                 end if;
22                 Put_Line (Integer'Image (B) & ".");
23             end Display_Equal;
24
25             Result : Boolean;
26         begin
27             case TC is
28                 when Equal_Chk =>
29                     for I in 0 .. 10 loop
30                         Result := Is_Equal (I, I);
31                         Display_Equal (I, I, Result);
32                     end loop;
33                 when Inequal_Chk =>
34                     for I in 0 .. 10 loop
35                         Result := Is_Equal (I, I - 1);
36                         Display_Equal (I, I - 1, Result);
37                     end loop;
38             end case;
39         end Check;
40
41     begin
42         if Argument_Count < 1 then
43             Put_Line ("ERROR: missing arguments! Exiting...");
44             return;
45         elsif Argument_Count > 1 then
46             Put_Line ("Ignoring additional arguments...");
47         end if;
48
49         Check (Test_Case_Index'Value (Argument (1)));
50     end Main;

```

118.4 States

Goal: write a procedure that displays the state of a machine.

Steps:

1. Complete the procedure `Display_State`.

Requirements:

1. The states can be set according to the following numbers:

Number	State
0	Off
1	On: Simple Processing
2	On: Advanced Processing

2. The procedure `Display_State` receives the number corresponding to a state and displays the state (indicated by the table above) as a user message.

Remarks:

1. You can use a case statement to implement this procedure.

Listing 2500: `display_state.ads`

```
1 procedure Display_State (State : Integer);
```

Listing 2501: `display_state.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_State (State : Integer) is
4 begin
5     null;
6 end Display_State;
```

Listing 2502: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_State;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Display_State (State);
19 end Main;
```

118.5 States #2

Goal: write a function that returns the state of a machine.

Steps:

1. Implement the function `Get_State`.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function `Get_State` must return the state as a string.

Remarks:

1. You can implement a function returning a string by simply using quotes in a return statement. For example:

Listing 2503: `get_hello.ads`

```
1 function Get_Hello return String;
```

Listing 2504: `get_hello.adb`

```
1 function Get_Hello return String is
2 begin
3   return "Hello";
4 end Get_Hello;
```

Listing 2505: `main.adb`

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Get_Hello;
3
4 procedure Main is
5   S : constant String := Get_Hello;
6 begin
7   Put_Line (S);
8 end Main;
```

2. You can reuse your previous implementation and replace it by a case expression.
 1. For values that do not correspond to a state, you can simply return an empty string `""`.

Listing 2506: `get_state.ads`

```
1 function Get_State (State : Integer) return String;
```

Listing 2507: `get_state.adb`

```
1 function Get_State (State : Integer) return String is
2 begin
3   return "";
4 end Get_State;
```

Listing 2508: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Get_State;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
```

(continues on next page)

(continued from previous page)

```

14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Put_Line (Get_State (State));
19 end Main;

```

118.6 States #3

Goal: implement an on/off indicator for a state machine.

Steps:

1. Implement the function `Is_On`.
2. Implement the procedure `Display_On_Off`.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function `Is_On` returns:
 - **True** if the machine is on;
 - otherwise, it returns **False**.
3. Procedure `Display_On_Off` displays the message
 - "On" if the machine is on, or
 - "Off" otherwise.
4. `Is_On` must be called in the implementation of `Display_On_Off`.

Remarks:

1. You can implement both subprograms using if expressions.

Listing 2509: `is_on.ads`

```

1 function Is_On (State : Integer) return Boolean;

```

Listing 2510: `is_on.adb`

```

1 function Is_On (State : Integer) return Boolean is
2 begin
3   return False;
4 end Is_On;

```

Listing 2511: `display_on_off.ads`

```

1 procedure Display_On_Off (State : Integer);

```

Listing 2512: `display_on_off.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Is_On;
3
4 procedure Display_On_Off (State : Integer) is
5 begin
6   Put_Line ("");
7 end Display_On_Off;

```

Listing 2513: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Display_On_Off;
5 with Is_On;
6
7 procedure Main is
8   State : Integer;
9 begin
10  if Argument_Count < 1 then
11    Put_Line ("ERROR: missing arguments! Exiting...");
12    return;
13  elsif Argument_Count > 1 then
14    Put_Line ("Ignoring additional arguments...");
15  end if;
16
17  State := Integer'Value (Argument (1));
18
19  Display_On_Off (State);
20  Put_Line (Boolean'Image (Is_On (State)));
21 end Main;
```

118.7 States #4

Goal: implement a procedure to update the state of a machine.

Steps:

1. Implement the procedure `Set_Next`.

Requirements:

1. Implement the same state machine as in the previous exercise.
2. Procedure `Set_Next` updates the machine's state with the next one in a *circular* manner:
 - In general, after a call to `Set_Next` for an integer variable `N` (`Set_Next (N)`), the new value of `N` must be the next number for that variable `N` (i.e., `N := N'Old + 1`).
 - In other words, a call to `Set_Next (N)` has the same effect as `N := N + 1`.
 - For example, after the statements `N := 1; Set_Next (N);`, we have that `N = 2`.
 - However, if the state is the last valid one for the machine (which, for this exercise, it's 2), the next state must be the first valid one (in this case: 0).
 - In other words, for `N := 2; Set_Next (N);`, we have that `N = 0`.

Remarks:

1. You can use an if expression to implement `Set_Next`.

Listing 2514: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 2515: set_next.adb

```
1 procedure Set_Next (State : in out Integer) is
2 begin
3     null;
4 end Set_Next;
```

Listing 2516: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Set_Next (State);
19     Put_Line (Integer'Image (State));
20 end Main;
```


MODULAR PROGRAMMING

119.1 Months

Goal: create a package to display the months of the year.

Steps:

1. Convert the Months procedure below to a package.
2. Create the specification and body of the Months package.

Requirements:

1. Months must contain the declaration of strings for each month of the year, which are stored in three-character constants based on the month's name.
 - For example, the string "January" is stored in the constant Jan. These strings are then used by the Display_Months procedure, which is also part of the Months package.

Remarks:

1. The goal of this exercise is to create the Months package.
 1. In the code below, Months is declared as a procedure.
 - Therefore, we need to *convert* it into a real package.
 2. You have to modify the procedure declaration and implementation in the code below, so that it becomes a package specification and a package body.

Listing 2517: months.ads

```
1  -- Create specification for Months package, which includes
2  -- the declaration of the Display_Months procedure.
3  --
4  procedure Months;
```

Listing 2518: months.adb

```
1  -- Create body of Months package, which includes
2  -- the implementation of the Display_Months procedure.
3  --
4  procedure Months is
5
6      procedure Display_Months is
7      begin
8          Put_Line ("Months:");
9          Put_Line ("- " & Jan);
10         Put_Line ("- " & Feb);
11         Put_Line ("- " & Mar);
12         Put_Line ("- " & Apr);
```

(continues on next page)

(continued from previous page)

```
13      Put_Line ("- " & May);
14      Put_Line ("- " & Jun);
15      Put_Line ("- " & Jul);
16      Put_Line ("- " & Aug);
17      Put_Line ("- " & Sep);
18      Put_Line ("- " & Oct);
19      Put_Line ("- " & Nov);
20      Put_Line ("- " & Dec);
21  end Display_Months;
22
23  begin
24      null;
25  end Months;
```

Listing 2519: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Months;            use Months;
5
6  procedure Main is
7
8      type Test_Case_Index is
9          (Months_Chk);
10
11      procedure Check (TC : Test_Case_Index) is
12      begin
13          case TC is
14              when Months_Chk =>
15                  Display_Months;
16          end case;
17      end Check;
18
19  begin
20      if Argument_Count < 1 then
21          Put_Line ("ERROR: missing arguments! Exiting...");
22          return;
23      elsif Argument_Count > 1 then
24          Put_Line ("Ignoring additional arguments...");
25      end if;
26
27      Check (Test_Case_Index'Value (Argument (1)));
28  end Main;
```

119.2 Operations

Goal: create a package to perform basic mathematical operations.

Steps:

1. Implement the Operations package.
 1. Declare and implement the Add function.
 2. Declare and implement the Subtract function.
 3. Declare and implement the Multiply: function.
 4. Declare and implement the Divide function.
2. Implement the Operations.Test package

1. Declare and implement the Display procedure.

Requirements:

1. Package Operations contains functions for each of the four basic mathematical operations for parameters of **Integer** type:
 1. Function Add performs the addition of A and B and returns the result;
 2. Function Subtract performs the subtraction of A and B and returns the result;
 3. Function Multiply performs the multiplication of A and B and returns the result;
 4. Function Divide performs the division of A and B and returns the result.
2. Package Operations.Test contains the test environment:
 1. Procedure Display must use the functions from the parent (Operations) package as indicated by the template in the code below.

Listing 2520: operations.ads

```

1 package Operations is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the functions mentioned above.
5   --
6
7 end Operations;
```

Listing 2521: operations.adb

```

1 package body Operations is
2
3   -- Create body of Operations package.
4   --
5
6 end Operations;
```

Listing 2522: operations-test.ads

```

1 package Operations.Test is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the Display procedure:
5   --
6   -- procedure Display (A, B : Integer);
7   --
8
9 end Operations.Test;
```

Listing 2523: operations-test.adb

```

1 package body Operations.Test is
2
3   -- Implement body of Operations.Test package.
4   --
5
6   procedure Display (A, B : Integer) is
7     A_Str : constant String := Integer'Image (A);
8     B_Str : constant String := Integer'Image (B);
9   begin
10    Put_Line ("Operations:");
11    Put_Line (A_Str & " + " & B_Str & " = ")
```

(continues on next page)

(continued from previous page)

```

12         & Integer'Image (Add (A, B))
13         & ",");
14     -- Use the line above as a template and add the rest of the
15     -- implementation for Subtract, Multiply and Divide.
16 end Display;
17
18 end Operations.Test;

```

Listing 2524: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Operations;
5  with Operations.Test; use Operations.Test;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Operations_Chk,
11          Operations_Display_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when Operations_Chk =>
17                 Put_Line ("Add (100, 2) = "
18                     & Integer'Image (Operations.Add (100, 2)));
19                 Put_Line ("Subtract (100, 2) = "
20                     & Integer'Image (Operations.Subtract (100, 2)));
21                 Put_Line ("Multiply (100, 2) = "
22                     & Integer'Image (Operations.Multiply (100, 2)));
23                 Put_Line ("Divide (100, 2) = "
24                     & Integer'Image (Operations.Divide (100, 2)));
25             when Operations_Display_Chk =>
26                 Display (10, 5);
27                 Display ( 1, 2);
28         end case;
29     end Check;
30
31 begin
32     if Argument_Count < 1 then
33         Put_Line ("ERROR: missing arguments! Exiting...");
34         return;
35     elsif Argument_Count > 1 then
36         Put_Line ("Ignoring additional arguments...");
37     end if;
38
39     Check (Test_Case_Index'Value (Argument (1)));
40 end Main;

```

STRONGLY TYPED LANGUAGE

120.1 Colors

Goal: create a package to represent HTML colors in hexadecimal form and its corresponding names.

Steps:

1. Implement the `Color_Types` package.
 1. Declare the `HTML_Color` enumeration.
 2. Declare the `Basic_HTML_Color` enumeration.
 3. Implement the `To_Integer` function.
 4. Implement the `To_HTML_Color` function.

Requirements:

1. Enumeration `HTML_Color` has the following colors:
 - Salmon
 - Firebrick
 - Red
 - Darkred
 - Lime
 - Forestgreen
 - Green
 - Darkgreen
 - Blue
 - Mediumblue
 - Darkblue
2. Enumeration `Basic_HTML_Color` has the following colors: Red, Green, Blue.
3. Function `To_Integer` converts from the `HTML_Color` type to the HTML color code — as integer values in hexadecimal notation.
 - You can find the HTML color codes in the table below.
4. Function `To_HTML_Color` converts from `Basic_HTML_Color` to `HTML_Color`.
5. This is the table to convert from an HTML color to a HTML color code in hexadecimal notation:

Color	HTML color code (hexa)
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

Remarks:

1. In order to express the hexadecimal values above in Ada, use the following syntax: `16#<hex_value>#` (e.g.: `16#FFFFFF#`).
2. For function `To_Integer`, you may use a **case** for this.

Listing 2525: color_types.ads

```
1 package Color_Types is
2
3   -- Include type declaration for HTML_Color!
4   --
5   -- type HTML_Color is [...]
6   --
7
8   -- Include function declaration for:
9   -- function To_Integer (C : HTML_Color) return Integer;
10
11  -- Include type declaration for Basic_HTML_Color!
12  --
13  -- type Basic_HTML_Color is [...]
14  --
15
16  -- Include function declaration for:
17  -- - Basic_HTML_Color => HTML_Color
18  --
19  -- function To_HTML_Color [...];
20  --
21 end Color_Types;
```

Listing 2526: color_types.adb

```
1 package body Color_Types is
2
3   -- Implement the conversion from HTML_Color to Integer here!
4   --
5   function To_Integer (C : HTML_Color) return Integer is
6   begin
7     -- Hint: use 'case' for the HTML colors;
8     --       use 16#...# for the hexadecimal values.
9     end To_Integer;
10
11  -- Implement the conversion from Basic_HTML_Color to HTML_Color here!
12  --
13  function To_HTML_Color [...] is
```

(continues on next page)

(continued from previous page)

```

14  --
15  end Color_Types;

```

Listing 2527: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Integer_Text_IO;
4
5  with Color_Types; use Color_Types;
6
7  procedure Main is
8      type Test_Case_Index is
9          (HTML_Color_Range,
10           HTML_Color_To_Integer,
11           Basic_HTML_Color_To_HTML_Color);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when HTML_Color_Range =>
17                 for I in HTML_Color'Range loop
18                     Put_Line (HTML_Color'Image (I));
19                 end loop;
20             when HTML_Color_To_Integer =>
21                 for I in HTML_Color'Range loop
22                     Ada.Integer_Text_IO.Put (Item => To_Integer (I),
23                                             Width => 6,
24                                             Base  => 16);
25                     New_Line;
26                 end loop;
27             when Basic_HTML_Color_To_HTML_Color =>
28                 for I in Basic_HTML_Color'Range loop
29                     Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30                 end loop;
31             end case;
32         end Check;
33
34     begin
35         if Argument_Count < 1 then
36             Put_Line ("ERROR: missing arguments! Exiting...");
37             return;
38         elsif Argument_Count > 1 then
39             Put_Line ("Ignoring additional arguments...");
40         end if;
41
42         Check (Test_Case_Index'Value (Argument (1)));
43     end Main;

```

120.2 Integers

Goal: implement a package with various integer types.

Steps:

1. Implement the Int_Types package.
 1. Declare the integer type I_100.
 2. Declare the modular type U_100.

3. Implement the `To_I_100` function to convert from the `U_100` type.
4. Implement the `To_U_100` function to convert from the `I_100` type.
5. Declare the derived type `D_50`.
6. Declare the subtype `S_50`.
7. Implement the `To_D_50` function to convert from the `I_100` type.
8. Implement the `To_S_50` function to convert from the `I_100` type.
9. Implement the `To_I_100` function to convert from the `D_50` type.

Requirements:

1. Types `I_100` and `U_100` have values between 0 and 100.
 1. Type `I_100` is an integer type.
 2. Type `U_100` is a modular type.
2. Function `To_I_100` converts from the `U_100` type to the `I_100` type.
3. Function `To_U_100` converts from the `I_100` type to the `U_100` type.
4. Types `D_50` and `S_50` have values between 10 and 50 and use `I_100` as a base type.
 1. `D_50` is a derived type.
 2. `S_50` is a subtype.
5. Function `To_D_50` converts from the `I_100` type to the `D_50` type.
6. Function `To_S_50` converts from the `I_100` type to the `S_50` type.
7. Functions `To_D_50` and `To_S_50` saturate the input values if they are out of range.
 - If the input is less than 10 the output should be 10.
 - If the input is greater than 50 the output should be 50.
8. Function `To_I_100` converts from the `D_50` type to the `I_100` type.

Remarks:

1. For the implementation of functions `To_D_50` and `To_S_50`, you may use the type attributes `D_50'First` and `D_50'Last`:
 1. `D_50'First` indicates the minimum value of the `D_50` type.
 2. `D_50'Last` indicates the maximum value of the `D_50` type.
 3. The same attributes are available for the `S_50` type (`S_50'First` and `S_50'Last`).
2. We could have implemented a function `To_I_100` as well to convert from `S_50` to `I_100`. However, we skip this here because explicit conversions are not needed for subtypes.

Listing 2528: `int_types.ads`

```
1 package Int_Types is
2
3   -- Include type declarations for I_100 and U_100!
4   --
5   type I_100 is [...];
6   type U_100 is [...];
7   --
8
9   function To_I_100 (V : U_100) return I_100;
10
11  function To_U_100 (V : I_100) return U_100;
```

(continues on next page)

(continued from previous page)

```

12
13  -- Include type declarations for D_50 and S_50!
14  --
15  -- [...] D_50 is [...]
16  -- [...] S_50 is [...]
17  --
18
19  function To_D_50 (V : I_100) return D_50;
20
21  function To_S_50 (V : I_100) return S_50;
22
23  function To_I_100 (V : D_50) return I_100;
24
25  end Int_Types;

```

Listing 2529: int_types.adb

```

1  package body Int_Types is
2
3      function To_I_100 (V : U_100) return I_100 is
4      begin
5          -- Implement the conversion from U_100 to I_100 here!
6          --
7          null;
8      end To_I_100;
9
10     function To_U_100 (V : I_100) return U_100 is
11     begin
12         -- Implement the conversion from I_100 to U_100 here!
13         --
14         null;
15     end To_U_100;
16
17     function To_D_50 (V : I_100) return D_50 is
18         Min : constant I_100 := I_100 (D_50'First);
19         Max : constant I_100 := I_100 (D_50'Last);
20     begin
21         -- Implement the conversion from I_100 to D_50 here!
22         --
23         -- Hint: using the constants above simplifies the checks needed for
24         --       this function.
25         --
26         null;
27     end To_D_50;
28
29     function To_S_50 (V : I_100) return S_50 is
30     begin
31         -- Implement the conversion from I_100 to S_50 here!
32         --
33         -- Remark: don't forget to verify whether an explicit conversion like
34         --       S_50 (V) is needed.
35         --
36         null;
37     end To_S_50;
38
39     function To_I_100 (V : D_50) return I_100 is
40     begin
41         -- Implement the conversion from I_100 to D_50 here!
42         --
43         -- Remark: don't forget to verify whether an explicit conversion like
44         --       I_100 (V) is needed.

```

(continues on next page)

(continued from previous page)

```

45     --
46     null;
47     end To_I_100;
48
49 end Int_Types;

```

Listing 2530: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7      package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8      package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9      package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11     use I_100_IO;
12     use U_100_IO;
13     use D_50_IO;
14
15     type Test_Case_Index is
16         (I_100_Range,
17          U_100_Range,
18          U_100_Wraparound,
19          U_100_To_I_100,
20          I_100_To_U_100,
21          D_50_Range,
22          S_50_Range,
23          I_100_To_D_50,
24          I_100_To_S_50,
25          D_50_To_I_100,
26          S_50_To_I_100);
27
28     procedure Check (TC : Test_Case_Index) is
29     begin
30         I_100_IO.Default_Width := 1;
31         U_100_IO.Default_Width := 1;
32         D_50_IO.Default_Width  := 1;
33
34         case TC is
35             when I_100_Range =>
36                 Put (I_100'First);
37                 New_Line;
38                 Put (I_100'Last);
39                 New_Line;
40             when U_100_Range =>
41                 Put (U_100'First);
42                 New_Line;
43                 Put (U_100'Last);
44                 New_Line;
45             when U_100_Wraparound =>
46                 Put (U_100'First - 1);
47                 New_Line;
48                 Put (U_100'Last + 1);
49                 New_Line;
50             when U_100_To_I_100 =>
51                 for I in U_100'Range loop
52                     I_100_IO.Put (To_I_100 (I));
53                     New_Line;

```

(continues on next page)

(continued from previous page)

```

54     end loop;
55     when I_100_To_U_100 =>
56         for I in I_100'Range loop
57             Put (To_U_100 (I));
58             New_Line;
59         end loop;
60     when D_50_Range =>
61         Put (D_50'First);
62         New_Line;
63         Put (D_50'Last);
64         New_Line;
65     when S_50_Range =>
66         Put (S_50'First);
67         New_Line;
68         Put (S_50'Last);
69         New_Line;
70     when I_100_To_D_50 =>
71         for I in I_100'Range loop
72             Put (To_D_50 (I));
73             New_Line;
74         end loop;
75     when I_100_To_S_50 =>
76         for I in I_100'Range loop
77             Put (To_S_50 (I));
78             New_Line;
79         end loop;
80     when D_50_To_I_100 =>
81         for I in D_50'Range loop
82             Put (To_I_100 (I));
83             New_Line;
84         end loop;
85     when S_50_To_I_100 =>
86         for I in S_50'Range loop
87             Put (I);
88             New_Line;
89         end loop;
90     end case;
91 end Check;
92
93 begin
94     if Argument_Count < 1 then
95         Put_Line ("ERROR: missing arguments! Exiting...");
96         return;
97     elsif Argument_Count > 1 then
98         Put_Line ("Ignoring additional arguments...");
99     end if;
100
101     Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

120.3 Temperatures

Goal: create a package to handle temperatures in Celsius and Kelvin.

Steps:

1. Implement the Temperature_Types package.
 1. Declare the Celsius type.
 2. Declare the Int_Celsius type.

3. Implement the `To_Celsius` function.
4. Implement the `To_Int_Celsius` function.
5. Declare the Kelvin type.
6. Implement the `To_Celsius` function to convert from the Kelvin type.
7. Implement the `To_Kelvin` function.

Requirements:

1. The custom floating-point types declared in `Temperature_Types` must use a precision of six digits.
2. Types `Celsius` and `Int_Celsius` are used for temperatures in Celsius:
 1. `Celsius` is a floating-point type with a range between -273.15 and 5504.85.
 2. `Int_Celsius` is an integer type with a range between -273 and 5505.
3. Functions `To_Celsius` and `To_Int_Celsius` are used for type conversion:
 1. `To_Celsius` converts from `Int_Celsius` to `Celsius` type.
 2. `To_Int_Celsius` converts from `Celsius` and `Int_Celsius` types:
4. Kelvin is a floating-point type for temperatures in Kelvin using a range between 0.0 and 5778.0.
5. The functions `To_Celsius` and `To_Kelvin` are used to convert between temperatures in Kelvin and Celsius.
 1. In order to convert temperatures in Celsius to Kelvin, you must use the formula $K = C + 273.15$, where:
 - K is the temperature in Kelvin, and
 - C is the temperature in Celsius.

Remarks:

1. When implementing the `To_Celsius` function for the `Int_Celsius` type:
 1. You'll need to check for the minimum and maximum values of the input values because of the slightly different ranges.
 2. You may use variables of floating-point type (**Float**) for intermediate values.
2. For the implementation of the functions `To_Celsius` and `To_Kelvin` (used for converting between Kelvin and Celsius), you may use a variable of floating-point type (**Float**) for intermediate values.

Listing 2531: `temperature_types.ads`

```
1 package Temperature_Types is
2
3   -- Include type declaration for Celsius!
4   --
5   -- Celsius is [...];
6   -- Int_Celsius is [...];
7   --
8
9   function To_Celsius (T : Int_Celsius) return Celsius;
10
11  function To_Int_Celsius (T : Celsius) return Int_Celsius;
12
13  -- Include type declaration for Kelvin!
14  --
15  -- type Kelvin is [...];
```

(continues on next page)

(continued from previous page)

```

16  --
17
18  -- Include function declarations for:
19  -- - Kelvin => Celsius
20  -- - Celsius => Kelvin
21  --
22  -- function To_Celsius [...];
23  -- function To_Kelvin  [...];
24  --
25  end Temperature_Types;

```

Listing 2532: temperature_types.adb

```

1  package body Temperature_Types is
2
3      function To_Celsius (T : Int_Celsius) return Celsius is
4      begin
5          null;
6      end To_Celsius;
7
8      function To_Int_Celsius (T : Celsius) return Int_Celsius is
9      begin
10         null;
11     end To_Int_Celsius;
12
13     -- Include function implementation for:
14     -- - Kelvin => Celsius
15     -- - Celsius => Kelvin
16     --
17     -- function To_Celsius [...] is
18     -- function To_Kelvin  [...] is
19     --
20  end Temperature_Types;

```

Listing 2533: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Temperature_Types; use Temperature_Types;
5
6  procedure Main is
7      package Celsius_IO    is new Ada.Text_IO.Float_IO (Celsius);
8      package Kelvin_IO     is new Ada.Text_IO.Float_IO (Kelvin);
9      package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
10
11      use Celsius_IO;
12      use Kelvin_IO;
13      use Int_Celsius_IO;
14
15      type Test_Case_Index is
16          (Celsius_Range,
17           Celsius_To_Int_Celsius,
18           Int_Celsius_To_Celsius,
19           Kelvin_To_Celsius,
20           Celsius_To_Kelvin);
21
22      procedure Check (TC : Test_Case_Index) is
23      begin
24          Celsius_IO.Default_Fore := 1;

```

(continues on next page)

(continued from previous page)

```

25     Kelvin_IO.Default_Fore := 1;
26     Int_Celsius_IO.Default_Width := 1;
27
28     case TC is
29         when Celsius_Range =>
30             Put (Celsius'First);
31             New_Line;
32             Put (Celsius'Last);
33             New_Line;
34         when Celsius_To_Int_Celsius =>
35             Put (To_Int_Celsius (Celsius'First));
36             New_Line;
37             Put (To_Int_Celsius (0.0));
38             New_Line;
39             Put (To_Int_Celsius (Celsius'Last));
40             New_Line;
41         when Int_Celsius_To_Celsius =>
42             Put (To_Celsius (Int_Celsius'First));
43             New_Line;
44             Put (To_Celsius (0));
45             New_Line;
46             Put (To_Celsius (Int_Celsius'Last));
47             New_Line;
48         when Kelvin_To_Celsius =>
49             Put (To_Celsius (Kelvin'First));
50             New_Line;
51             Put (To_Celsius (0));
52             New_Line;
53             Put (To_Celsius (Kelvin'Last));
54             New_Line;
55         when Celsius_To_Kelvin =>
56             Put (To_Kelvin (Celsius'First));
57             New_Line;
58             Put (To_Kelvin (Celsius'Last));
59             New_Line;
60     end case;
61 end Check;
62
63 begin
64     if Argument_Count < 1 then
65         Put_Line ("ERROR: missing arguments! Exiting...");
66         return;
67     elsif Argument_Count > 1 then
68         Put_Line ("Ignoring additional arguments...");
69     end if;
70
71     Check (Test_Case_Index'Value (Argument (1)));
72 end Main;

```

RECORDS

121.1 Directions

Goal: create a package that handles directions and geometric angles.

Steps:

1. Implement the Directions package.
 1. Declare the Ext_Angle record.
 2. Implement the Display procedure.
 3. Implement the To_Ext_Angle function.

Requirements:

1. Record Ext_Angle stores information about the extended angle (see remark about *extended angles* below).
2. Procedure Display displays information about the extended angle.
 1. You should use the implementation that has been commented out (see code below) as a starting point.
3. Function To_Ext_Angle converts a simple angle value to an extended angle (Ext_Angle type).

Remarks:

1. We make use of the algorithm implemented in the Check_Direction procedure ([chapter on imperative language](#) (page 9)).
2. For the sake of this exercise, we use the concept of *extended angles*. This includes the actual geometric angle and the corresponding direction (North, South, Northwest, and so on).

Listing 2534: directions.ads

```
1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northeast,
8      East,
9      Southeast,
10     South,
11     Southwest,
12     West,
13     Northwest);
14
```

(continues on next page)

(continued from previous page)

```

15  function To_Direction (N: Angle_Mod) return Direction;
16
17  -- Include type declaration for Ext_Angle record type:
18  --
19  -- NOTE: Use the Angle_Mod and Direction types declared above!
20  --
21  -- type Ext_Angle is [...]
22  --
23
24  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
25
26  procedure Display (N : Ext_Angle);
27
28  end Directions;

```

Listing 2535: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          -- Uncomment the code below and fill the missing elements
8          --
9          -- Put_Line ("Angle: "
10             & Angle_Mod'Image (____)
11             & " => "
12             & Direction'Image (____)
13             & ".");
14          null;
15      end Display;
16
17      function To_Direction (N : Angle_Mod) return Direction is
18      begin
19          case N is
20              when 0      => return North;
21              when 1 .. 89 => return Northeast;
22              when 90     => return East;
23              when 91 .. 179 => return Southeast;
24              when 180    => return South;
25              when 181 .. 269 => return Southwest;
26              when 270    => return West;
27              when 271 .. 359 => return Northwest;
28          end case;
29      end To_Direction;
30
31      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
32      begin
33          -- Implement the conversion from Angle_Mod to Ext_Angle here!
34          --
35          -- Hint: you can use a return statement and an aggregate.
36          --
37          null;
38      end To_Ext_Angle;
39
40  end Directions;

```

Listing 2536: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Directions;       use Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Direction_Chk =>
14             Display (To_Ext_Angle (0));
15             Display (To_Ext_Angle (30));
16             Display (To_Ext_Angle (45));
17             Display (To_Ext_Angle (90));
18             Display (To_Ext_Angle (91));
19             Display (To_Ext_Angle (120));
20             Display (To_Ext_Angle (180));
21             Display (To_Ext_Angle (250));
22             Display (To_Ext_Angle (270));
23         end case;
24     end Check;
25
26 begin
27     if Argument_Count < 1 then
28         Put_Line ("ERROR: missing arguments! Exiting...");
29         return;
30     elsif Argument_Count > 1 then
31         Put_Line ("Ignoring additional arguments...");
32     end if;
33
34     Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

121.2 Colors

Goal: create a package to represent HTML colors in RGB format using the hexadecimal form.

Steps:

1. Implement the Color_Types package.
 1. Declare the RGB record.
 2. Implement the To_RGB function.
 3. Implement the Image function for the RGB type.

Requirements:

1. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

2. The hexadecimal information of each HTML color can be mapped to three color elements: red, green and blue.
 1. Each color element has a value between 0 and 255, or 00 and FF in hexadecimal.
 2. For example, for the color *salmon*, the hexadecimal value of the color elements are:
 - red = FA,
 - green = 80, and
 - blue = 72.
3. Record RGB stores information about HTML colors in RGB format, so that we can retrieve the individual color elements.
4. Function To_RGB converts from the HTML_Color enumeration to the RGB type based on the information from the table above.
5. Function Image returns a string representation of the RGB type in this format:
 - "(Red => 16#..#, Green => 16#...#, Blue => 16#...#)"

Remarks:

1. We use the exercise on HTML colors from the previous lab on *Strongly typed language* (page 2175) as a starting point.

Listing 2537: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,
```

(continues on next page)

(continued from previous page)

```

20     Green,
21     Blue);
22
23     function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25     subtype Int_Color is Integer range 0 .. 255;
26
27     -- Replace type declaration for RGB record below
28     --
29     -- - NOTE: Use the Int_Color type declared above!
30     --
31     -- type RGB is [...]
32     --
33     type RGB is null record;
34
35     function To_RGB (C : HTML_Color) return RGB;
36
37     function Image (C : RGB) return String;
38
39 end Color_Types;

```

Listing 2538: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_Integer (C : HTML_Color) return Integer is
6      begin
7          case C is
8              when Salmon    => return 16#FA8072#;
9              when Firebrick => return 16#B22222#;
10             when Red        => return 16#FF0000#;
11             when Darkred    => return 16#8B0000#;
12             when Lime       => return 16#00FF00#;
13             when Forestgreen => return 16#228B22#;
14             when Green      => return 16#008000#;
15             when Darkgreen  => return 16#006400#;
16             when Blue       => return 16#0000FF#;
17             when Mediumblue => return 16#0000CD#;
18             when Darkblue   => return 16#00008B#;
19          end case;
20
21      end To_Integer;
22
23      function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24      begin
25          case C is
26              when Red    => return Red;
27              when Green  => return Green;
28              when Blue   => return Blue;
29          end case;
30      end To_HTML_Color;
31
32      function To_RGB (C : HTML_Color) return RGB is
33      begin
34          -- Implement the conversion from HTML_Color to RGB here!
35          --
36          return (null record);
37      end To_RGB;
38

```

(continues on next page)

(continued from previous page)

```

39  function Image (C : RGB) return String is
40      subtype Str_Range is Integer range 1 .. 10;
41      SR : String (Str_Range);
42      SG : String (Str_Range);
43      SB : String (Str_Range);
44  begin
45      -- Replace argument in the calls to Put below
46      -- with the missing elements (red, green, blue)
47      -- from the RGB record
48      --
49      Ada.Integer_Text_IO.Put (To   => SR,
50                              Item  => 0,    -- REPLACE!
51                              Base  => 16);
52      Ada.Integer_Text_IO.Put (To   => SG,
53                              Item  => 0,    -- REPLACE!
54                              Base  => 16);
55      Ada.Integer_Text_IO.Put (To   => SB,
56                              Item  => 0,    -- REPLACE!
57                              Base  => 16);
58      return ("(Red => " & SR
59              & ", Green => " & SG
60              & ", Blue => " & SB
61              & ")");
62  end Image;
63
64  end Color_Types;

```

Listing 2539: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_To_RGB);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when HTML_Color_To_RGB =>
14                 for I in HTML_Color'Range loop
15                     Put_Line (HTML_Color'Image (I) & " => "
16                             & Image (To_RGB (I)) & ".");
17                 end loop;
18             end case;
19     end Check;
20
21     begin
22         if Argument_Count < 1 then
23             Put_Line ("ERROR: missing arguments! Exiting...");
24             return;
25         elsif Argument_Count > 1 then
26             Put_Line ("Ignoring additional arguments...");
27         end if;
28
29         Check (Test_Case_Index'Value (Argument (1)));
30     end Main;

```

121.3 Inventory

Goal: create a simplified inventory system for a store to enter items and keep track of assets.

Steps:

1. Implement the Inventory_Pkg package.
 1. Declare the Item record.
 2. Implement the Init function.
 3. Implement the Add procedure.

Requirements:

1. Record Item collects information about products from the store.
 1. To keep it simple, this record only contains the name, quantity and price of each item.
 2. The record components are:
 - Name of Item_Name type;
 - Quantity of **Natural** type;
 - Price of **Float** type.
2. Function Init returns an initialized item (of Item type).
 1. Function Init must also display the item name by calling the To_String function for the Item_Name type.
 - This is already implemented in the code below.
3. Procedure Add adds an item to the assets.
 1. Since we want to keep track of the assets, the implementation must accumulate the total value of each item's inventory, the result of multiplying the item quantity and its price.

Listing 2540: inventory_pkg.ads

```

1 package Inventory_Pkg is
2
3   type Item_Name is
4     (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6   function To_String (I : Item_Name) return String;
7
8   -- Replace type declaration for Item record:
9   --
10  type Item is null record;
11
12  function Init (Name      : Item_Name;
13                Quantity  : Natural;
14                Price     : Float) return Item;
15
16  procedure Add (Assets : in out Float;
17               I       : Item);
18
19 end Inventory_Pkg;
```

Listing 2541: inventory_pkg.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Inventory_Pkg is
4
5      function To_String (I : Item_Name) return String is
6      begin
7          case I is
8              when Ballpoint_Pen      => return "Ballpoint Pen";
9              when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10             when Feather_Quill_Pen   => return "Feather Quill Pen";
11          end case;
12      end To_String;
13
14      function Init (Name      : Item_Name;
15                   Quantity  : Natural;
16                   Price     : Float) return Item is
17      begin
18          Put_Line ("Item: " & To_String (Name) & ".");
19
20          -- Replace return statement with the actual record initialization!
21          --
22          return (null record);
23      end Init;
24
25      procedure Add (Assets : in out Float;
26                   I       : Item) is
27      begin
28          -- Implement the function that adds an item to the inventory here!
29          --
30          null;
31      end Add;
32
33  end Inventory_Pkg;

```

Listing 2542: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Inventory_Pkg;     use Inventory_Pkg;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42);
9
10     type Test_Case_Index is
11         (Inventory_Chk);
12
13     procedure Display (Assets : Float) is
14         package F_IO is new Ada.Text_IO.Float_IO (Float);
15
16         use F_IO;
17     begin
18         Put ("Assets: $");
19         Put (Assets, 1, 2, 0);
20         Put (".");
21         New_Line;
22     end Display;
23

```

(continues on next page)

(continued from previous page)

```

24  procedure Check (TC : Test_Case_Index) is
25      I      : Item;
26      Assets : Float := 0.0;
27
28      -- Please ignore the following three lines!
29      pragma Warnings (Off, "default initialization");
30      for Assets'Address use F'Address;
31      pragma Warnings (On, "default initialization");
32  begin
33      case TC is
34      when Inventory_Chk =>
35          I := Init (Ballpoint_Pen,      185,  0.15);
36          Add (Assets, I);
37          Display (Assets);
38
39          I := Init (Oil_Based_Pen_Marker, 100,  9.0);
40          Add (Assets, I);
41          Display (Assets);
42
43          I := Init (Feather_Quill_Pen,    2, 40.0);
44          Add (Assets, I);
45          Display (Assets);
46      end case;
47  end Check;
48
49  begin
50      if Argument_Count < 1 then
51          Put_Line ("ERROR: missing arguments! Exiting...");
52          return;
53      elsif Argument_Count > 1 then
54          Put_Line ("Ignoring additional arguments...");
55      end if;
56
57      Check (Test_Case_Index'Value (Argument (1)));
58  end Main;

```


ARRAYS

122.1 Constrained Array

Goal: declare a constrained array and implement operations on it.

Steps:

1. Implement the `Constrained_Arrays` package.
 1. Declare the range type `My_Index`.
 2. Declare the array type `My_Array`.
 3. Declare and implement the `Init` function.
 4. Declare and implement the `Double` procedure.
 5. Declare and implement the `First_Elem` function.
 6. Declare and implement the `Last_Elem` function.
 7. Declare and implement the `Length` function.
 8. Declare the object `A` of `My_Array` type.

Requirements:

1. Range type `My_Index` has a range from 1 to 10.
2. `My_Array` is a constrained array of **Integer** type.
 1. It must make use of the `My_Index` type.
 2. It is therefore limited to 10 elements.
3. Function `Init` returns an array where each element is initialized with the corresponding index.
4. Procedure `Double` doubles the value of each element of an array.
5. Function `First_Elem` returns the first element of the array.
6. Function `Last_Elem` returns the last element of the array.
7. Function `Length` returns the length of the array.
8. Object `A` of `My_Array` type is initialized with:
 1. the values 1 and 2 for the first two elements, and
 2. 42 for all other elements.

Listing 2543: constrained_arrays.ads

```

1 package Constrained_Arrays is
2
3     -- Complete the type and subprogram declarations:
4     --
5     -- type My_Index is [...]
6     --
7     -- type My_Array is [...]
8     --
9     -- function Init ...
10    --
11    -- procedure Double ...
12    --
13    -- function First_Elem ...
14    --
15    -- function Last_Elem ...
16    --
17    -- function Length ...
18    --
19    -- A : ...
20
21 end Constrained_Arrays;
```

Listing 2544: constrained_arrays.adb

```

1 package body Constrained_Arrays is
2
3     -- Create the implementation of the subprograms!
4     --
5
6 end Constrained_Arrays;
```

Listing 2545: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3
4 with Constrained_Arrays; use Constrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Range_Chk,
9          Array_Range_Chk,
10         A_Obj_Chk,
11         Init_Chk,
12         Double_Chk,
13         First_Elem_Chk,
14         Last_Elem_Chk,
15         Length_Chk);
16
17     procedure Check (TC : Test_Case_Index) is
18         AA : My_Array;
19
20         procedure Display (A : My_Array) is
21             begin
22                 for I in A'Range loop
23                     Put_Line (Integer'Image (A (I)));
24                 end loop;
25             end Display;
26
27     procedure Local_Init (A : in out My_Array) is
```

(continues on next page)

(continued from previous page)

```

28     begin
29         A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30     end Local_Init;
31 begin
32     case TC is
33     when Range_Chk =>
34         for I in My_Index loop
35             Put_Line (My_Index'Image (I));
36         end loop;
37     when Array_Range_Chk =>
38         for I in My_Array'Range loop
39             Put_Line (My_Index'Image (I));
40         end loop;
41     when A_Obj_Chk =>
42         Display (A);
43     when Init_Chk =>
44         AA := Init;
45         Display (AA);
46     when Double_Chk =>
47         Local_Init (AA);
48         Double (AA);
49         Display (AA);
50     when First_Elem_Chk =>
51         Local_Init (AA);
52         Put_Line (Integer'Image (First_Elem (AA)));
53     when Last_Elem_Chk =>
54         Local_Init (AA);
55         Put_Line (Integer'Image (Last_Elem (AA)));
56     when Length_Chk =>
57         Put_Line (Integer'Image (Length (AA)));
58     end case;
59 end Check;
60
61 begin
62     if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");
64         return;
65     elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");
67     end if;
68
69     Check (Test_Case_Index'Value (Argument (1)));
70 end Main;

```

122.2 Colors: Lookup-Table

Goal: rewrite a package to represent HTML colors in RGB format using a lookup table.

Steps:

1. Implement the Color_Types package.
 1. Declare the array type HTML_Color_RGB.
 2. Declare the To_RGB_Lookup_Table object and initialize it.
 3. Adapt the implementation of the To_RGB function.

Requirements:

1. Array type HTML_Color_RGB is used for the table.
2. The To_RGB_Lookup_Table object of HTML_Color_RGB type contains the lookup table.

- This table must be implemented as an array of constant values.
3. The implementation of the To_RGB function must use the To_RGB_Lookup_Table object.

Remarks:

1. This exercise is based on the HTML colors exercise from a previous lab ([Records](#) (page 2185)).
2. In the previous implementation, you could use a **case** statement to implement the To_RGB function. Here, you must rewrite the function using a look-up table.
 1. The implementation of the To_RGB function below includes the case statement as commented-out code. You can use this as your starting point: you just need to copy it and convert the case statement to an array declaration.
 1. Don't use a case statement to implement the To_RGB function. Instead, write code that accesses To_RGB_Lookup_Table to get the correct value.
3. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

Listing 2546: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   subtype Int_Color is Integer range 0 .. 255;
17
18   type RGB is record
19     Red   : Int_Color;
20     Green : Int_Color;
21     Blue  : Int_Color;
22   end record;
23

```

(continues on next page)

(continued from previous page)

```

24  function To_RGB (C : HTML_Color) return RGB;
25
26  function Image (C : RGB) return String;
27
28  -- Declare array type for lookup table here:
29  --
30  -- type HTML_Color_RGB is ...
31
32  -- Declare lookup table here:
33  --
34  -- To_RGB_Lookup_Table : ...
35
36  end Color_Types;

```

Listing 2547: color_types.adb

```

1  with Ada.Integer_Text_IO;
2  package body Color_Types is
3
4      function To_RGB (C : HTML_Color) return RGB is
5      begin
6          -- Implement To_RGB using To_RGB_Lookup_Table
7          return (0, 0, 0);
8
9          -- Use the code below from the previous version of the To_RGB
10         -- function to declare the To_RGB_Lookup_Table:
11         --
12         -- case C is
13         --     when Salmon      => return (16#FA#, 16#80#, 16#72#);
14         --     when Firebrick   => return (16#B2#, 16#22#, 16#22#);
15         --     when Red         => return (16#FF#, 16#00#, 16#00#);
16         --     when Darkred     => return (16#8B#, 16#00#, 16#00#);
17         --     when Lime        => return (16#00#, 16#FF#, 16#00#);
18         --     when Forestgreen => return (16#22#, 16#8B#, 16#22#);
19         --     when Green       => return (16#00#, 16#80#, 16#00#);
20         --     when Darkgreen   => return (16#00#, 16#64#, 16#00#);
21         --     when Blue        => return (16#00#, 16#00#, 16#FF#);
22         --     when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
23         --     when Darkblue    => return (16#00#, 16#00#, 16#8B#);
24         -- end case;
25
26     end To_RGB;
27
28     function Image (C : RGB) return String is
29     subtype Str_Range is Integer range 1 .. 10;
30     SR : String (Str_Range);
31     SG : String (Str_Range);
32     SB : String (Str_Range);
33     begin
34         Ada.Integer_Text_IO.Put (To    => SR,
35                                Item   => C.Red,
36                                Base    => 16);
37         Ada.Integer_Text_IO.Put (To    => SG,
38                                Item   => C.Green,
39                                Base    => 16);
40         Ada.Integer_Text_IO.Put (To    => SB,
41                                Item   => C.Blue,
42                                Base    => 16);
43
44         return ("(Red => " & SR
45                & ", Green => " & SG
46                & ", Blue => " & SB

```

(continues on next page)

(continued from previous page)

```

46         &"");
47     end Image;
48
49 end Color_Types;

```

Listing 2548: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Color_Types;           use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Color_Table_Chk,
9           HTML_Color_To_Integer_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Color_Table_Chk =>
15                 Put_Line ("Size of HTML_Color_RGB: "
16                     & Integer'Image (HTML_Color_RGB'Length));
17                 Put_Line ("Firebrick: "
18                     & Image (To_RGB_Lookup_Table (Firebrick)));
19             when HTML_Color_To_Integer_Chk =>
20                 for I in HTML_Color'Range loop
21                     Put_Line (HTML_Color'Image (I) & " => "
22                         & Image (To_RGB (I)) & ".");
23                 end loop;
24             end case;
25     end Check;
26
27 begin
28     if Argument_Count < 1 then
29         Put_Line ("ERROR: missing arguments! Exiting...");
30         return;
31     elsif Argument_Count > 1 then
32         Put_Line ("Ignoring additional arguments...");
33     end if;
34
35     Check (Test_Case_Index'Value (Argument (1)));
36 end Main;

```

122.3 Unconstrained Array

Goal: declare an unconstrained array and implement operations on it.

Steps:

1. Implement the Unconstrained_Arrays package.
 1. Declare the My_Array type.
 2. Declare and implement the Init procedure.
 3. Declare and implement the Init function.
 4. Declare and implement the Double procedure.
 5. Declare and implement the Diff_Prev_Elem function.

Requirements:

1. My_Array is an unconstrained array (with a **Positive** range) of **Integer** elements.
2. Procedure Init initializes each element with the index starting with the last one.
 - For example, for an array of 3 elements where the index of the first element is 1 (My_Array (1 .. 3)), the values of these elements after a call to Init must be (3, 2, 1).
3. Function Init returns an array based on the length L and start index I provided to the Init function.
 1. I indicates the index of the first element of the array.
 2. L indicates the length of the array.
 3. Both I and L must be positive.
 4. This is its declaration: **function** Init (I, L : Positive) **return** My_Array;.
 5. You must initialize the elements of the array in the same manner as for the Init procedure described above.
4. Procedure Double doubles each element of an array.
5. Function Diff_Prev_Elem returns — for each element of an input array A — an array with the difference between an element of array A and the previous element.
 1. For the first element, the difference must be zero.
 2. For example:
 - **INPUT:** (2, 5, 15)
 - **RETURN** of Diff_Prev_Elem: (0, 3, 10), where
 - 0 is the constant difference for the first element;
 - $5 - 2 = 3$ is the difference between the second and the first elements of the input array;
 - $15 - 5 = 10$ is the difference between the third and the second elements of the input array.

Remarks:

1. For an array A, you can retrieve the index of the last element with the attribute 'Last'.
 1. For example: Y : **Positive** := A'Last;
 2. This can be useful during the implementation of procedure Init.
2. For the implementation of the Init function, you can call the Init procedure to initialize the elements. By doing this, you avoid code duplication.
3. Some hints about attributes:
 1. You can use the range attribute (A'Range) to retrieve the range of an array A.
 2. You can also use the range attribute in the declaration of another array (e.g.: B : My_Array (A'Range)).
 3. Alternatively, you can use the A'First and A'Last attributes in an array declaration.

Listing 2549: unconstrained_arrays.ads

```

1 package Unconstrained_Arrays is
2
3   -- Complete the type and subprogram declarations:

```

(continues on next page)

(continued from previous page)

```

4      --
5      --  type My_Array is ...;
6      --
7      --  procedure Init ...;
8
9      function Init (I, L : Positive) return My_Array;
10
11     --  procedure Double ...;
12     --
13     --  function Diff_Prev_Elem ...;
14
15 end Unconstrained_Arrays;
```

Listing 2550: unconstrained_arrays.adb

```

1 package body Unconstrained_Arrays is
2
3     --  Implement the subprograms:
4     --
5
6     --  procedure Init is...
7
8     --  function Init (L : Positive) return My_Array is...
9
10    --  procedure Double ... is...
11
12    --  function Diff_Prev_Elem ... is...
13
14 end Unconstrained_Arrays;
```

Listing 2551: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Init_Chk,
9          Init_Proc_Chk,
10         Double_Chk,
11         Diff_Prev_Chk,
12         Diff_Prev_Single_Chk);
13
14     procedure Check (TC : Test_Case_Index) is
15         AA : My_Array (1 .. 5);
16         AB : My_Array (5 .. 9);
17
18         procedure Display (A : My_Array) is
19             begin
20                 for I in A'Range loop
21                     Put_Line (Integer'Image (A (I)));
22                 end loop;
23             end Display;
24
25         procedure Local_Init (A : in out My_Array) is
26             begin
27                 A := (1, 2, 5, 10, -10);
28             end Local_Init;
```

(continues on next page)

(continued from previous page)

```

29
30   begin
31       case TC is
32       when Init_Chk =>
33           AA := Init (AA'First, AA'Length);
34           AB := Init (AB'First, AB'Length);
35           Display (AA);
36           Display (AB);
37       when Init_Proc_Chk =>
38           Init (AA);
39           Init (AB);
40           Display (AA);
41           Display (AB);
42       when Double_Chk =>
43           Local_Init (AB);
44           Double (AB);
45           Display (AB);
46       when Diff_Prev_Chk =>
47           Local_Init (AB);
48           AB := Diff_Prev_Elem (AB);
49           Display (AB);
50       when Diff_Prev_Single_Chk =>
51           declare
52               A1 : My_Array (1 .. 1) := (1 => 42);
53           begin
54               A1 := Diff_Prev_Elem (A1);
55               Display (A1);
56           end;
57       end case;
58   end Check;
59
60   begin
61       if Argument_Count < 1 then
62           Put_Line ("ERROR: missing arguments! Exiting...");
63           return;
64       elsif Argument_Count > 1 then
65           Put_Line ("Ignoring additional arguments...");
66       end if;
67
68       Check (Test_Case_Index'Value (Argument (1)));
69   end Main;

```

122.4 Product info

Goal: create a system to keep track of quantities and prices of products.

Steps:

1. Implement the Product_Info_Pkg package.
 1. Declare the array type Product_Infos.
 2. Declare the array type Currency_Array.
 3. Implement the Total procedure.
 4. Implement the Total function returning an array of Currency_Array type.
 5. Implement the Total function returning a single value of Currency type.

Requirements:

1. Quantity of an individual product is represented by the Quantity subtype.

2. Price of an individual product is represented by the Currency subtype.
3. Record type Product_Info deals with information for various products.
4. Array type Product_Infos is used to represent a list of products.
5. Array type Currency_Array is used to represent a list of total values of individual products (see more details below).
6. Procedure Total receives an input array of products.
 1. It outputs an array with the total value of each product using the Currency_Array type.
 2. The total value of an individual product is calculated by multiplying the quantity for this product by its price.
7. Function Total returns an array of Currency_Array type.
 1. This function has the same purpose as the procedure Total.
 2. The difference is that the function returns an array instead of providing this array as an output parameter.
8. The second function Total returns a single value of Currency type.
 1. This function receives an array of products.
 2. It returns a single value corresponding to the total value for all products in the system.

Remarks:

1. You can use Currency (Q) to convert from an element Q of Quantity type to the Currency type.
 1. As you might remember, Ada requires an explicit conversion in calculations where variables of both integer and floating-point types are used.
 2. In our case, the Quantity subtype is based on the **Integer** type and the Currency subtype is based on the **Float** type, so a conversion is necessary in calculations using those types.

Listing 2552: product_info_pkg.ads

```
1 package Product_Info_Pkg is
2
3     subtype Quantity is Natural;
4
5     subtype Currency is Float;
6
7     type Product_Info is record
8         Units : Quantity;
9         Price : Currency;
10    end record;
11
12    -- Complete the type declarations:
13    --
14    -- type Product_Infos is ...
15    --
16    -- type Currency_Array is ...
17
18    procedure Total (P : Product_Infos;
19                   Tot : out Currency_Array);
20
21    function Total (P : Product_Infos) return Currency_Array;
22
23    function Total (P : Product_Infos) return Currency;
```

(continues on next page)

(continued from previous page)

```

24
25 end Product_Info_Pkg;

```

Listing 2553: product_info_pkg.adb

```

1 package body Product_Info_Pkg is
2
3   -- Complete the subprogram implementations:
4   --
5
6   -- procedure Total (P : Product_Infos;
7   --                  Tot : out Currency_Array) is ...
8
9   -- function Total (P : Product_Infos) return Currency_Array is ...
10
11  -- function Total (P : Product_Infos) return Currency is ...
12
13 end Product_Info_Pkg;

```

Listing 2554: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Product_Info_Pkg; use Product_Info_Pkg;
5
6 procedure Main is
7   package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9   type Test_Case_Index is
10     (Total_Func_Chk,
11      Total_Proc_Chk,
12      Total_Value_Chk);
13
14   procedure Check (TC : Test_Case_Index) is
15     subtype Test_Range is Positive range 1 .. 5;
16
17     P : Product_Infos (Test_Range);
18     Tots : Currency_Array (Test_Range);
19     Tot : Currency;
20
21     procedure Display (Tots : Currency_Array) is
22     begin
23       for I in Tots'Range loop
24         Currency_IO.Put (Tots (I));
25         New_Line;
26       end loop;
27     end Display;
28
29     procedure Local_Init (P : in out Product_Infos) is
30     begin
31       P := ((1, 0.5),
32            (2, 10.0),
33            (5, 40.0),
34            (10, 10.0),
35            (10, 20.0));
36     end Local_Init;
37
38   begin
39     Currency_IO.Default_Fore := 1;

```

(continues on next page)

(continued from previous page)

```

40     Currency_IO.Default_Aft := 2;
41     Currency_IO.Default_Exp := 0;
42
43     case TC is
44     when Total_Func_Chk =>
45         Local_Init (P);
46         Tots := Total (P);
47         Display (Tots);
48     when Total_Proc_Chk =>
49         Local_Init (P);
50         Total (P, Tots);
51         Display (Tots);
52     when Total_Value_Chk =>
53         Local_Init (P);
54         Tot := Total (P);
55         Currency_IO.Put (Tot);
56         New_Line;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

122.5 String_10

Goal: work with constrained string types.

Steps:

1. Implement the Strings_10 package.
 1. Declare the String_10 type.
 2. Implement the To_String_10 function.

Requirements:

1. The constrained string type String_10 is an array of ten characters.
2. Function To_String_10 returns constrained strings of String_10 type based on an input parameter of **String** type.
 - For strings that are more than 10 characters, omit everything after the 11th character.
 - For strings that are fewer than 10 characters, pad the string with ' ' characters until it is 10 characters.

Remarks:

1. Declaring String_10 as a subtype of **String** is the easiest way.
 - You may declare it as a new type as well. However, this requires some adaptations in the Main test procedure.
2. You can use **Integer'Min** to calculate the minimum of two integer values.

Listing 2555: strings_10.ads

```

1 package Strings_10 is
2
3     -- Complete the type and subprogram declarations:
4     --
5
6     -- subtype String_10 is ...;
7
8     -- Using "type String_10 is..." is possible, too. However, it
9     -- requires a custom Put_Line procedure that is called in Main:
10    -- procedure Put_Line (S : String_10);
11
12    -- function To_String_10 ...;
13
14 end Strings_10;

```

Listing 2556: strings_10.adb

```

1 package body Strings_10 is
2
3     -- Complete the subprogram declaration and implementation:
4     --
5     -- function To_String_10 ... is
6
7 end Strings_10;

```

Listing 2557: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3
4 with Strings_10;          use Strings_10;
5
6 procedure Main is
7     type Test_Case_Index is
8         (String_10_Long_Chk,
9          String_10_Short_Chk);
10
11    procedure Check (TC : Test_Case_Index) is
12        SL : constant String := "And this is a long string just for testing...";
13        SS : constant String := "Hey!";
14        S_10 : String_10;
15
16    begin
17        case TC is
18            when String_10_Long_Chk =>
19                S_10 := To_String_10 (SL);
20                Put_Line (String (S_10));
21            when String_10_Short_Chk =>
22                S_10 := (others => ' ');
23                S_10 := To_String_10 (SS);
24                Put_Line (String (S_10));
25            end case;
26        end Check;
27
28    begin
29        if Argument_Count < 1 then
30            Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
31            return;
32        elsif Argument_Count > 1 then

```

(continues on next page)

(continued from previous page)

```
33     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
34   end if;
35
36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;
```

122.6 List of Names

Goal: create a system for a list of names and ages.

Steps:

1. Implement the Names_Ages package.
 1. Declare the People_Array array type.
 2. Complete the declaration of the People record type with the People_A element of People_Array type.
 3. Implement the Add procedure.
 4. Implement the Reset procedure.
 5. Implement the Get function.
 6. Implement the Update procedure.
 7. Implement the Display procedure.

Requirements:

1. Each person is represented by the Person type, which is a record containing the name and the age of that person.
2. People_Array is an unconstrained array of Person type with a positive range.
3. The Max_People constant is set to 10.
4. Record type People contains:
 1. The People_A element of People_Array type.
 2. This array must be constrained by the Max_People constant.
5. Procedure Add adds a person to the list.
 1. By default, the age of this person is set to zero in this procedure.
6. Procedure Reset resets the list.
7. Function Get retrieves the age of a person from the list.
8. Procedure Update updates the age of a person in the list.
9. Procedure Display shows the complete list using the following format:
 1. The first line must be LIST OF NAMES:. It is followed by the name and age of each person in the next lines.
 2. For each person on the list, the procedure must display the information in the following format:

```
NAME: XXXX
AGE: YY
```

Remarks:

1. In the implementation of procedure `Add`, you may use an index to indicate the last valid position in the array — see `Last_Valid` in the code below.
2. In the implementation of procedure `Display`, you should use the `Trim` function from the `Ada.Strings.Fixed` package to format the person's name — for example: `Trim (P.Name, Right)`.
3. You may need the `Integer'Min` (A, B) and the `Integer'Max` (A, B) functions to get the minimum and maximum values in a comparison between two integer values A and B.
4. Fixed-length strings can be initialized with whitespaces using the `others` syntax. For example: `S : String_10 := (others => ' ');`
5. You may implement additional subprograms to deal with other types declared in the `Names_Ages` package below, such as the `Name_Type` and the `Person` type.
 1. For example, a function `To_Name_Type` to convert from `String` to `Name_Type` might be useful.
 2. Take a moment to reflect on which additional subprograms could be useful as well.

Listing 2558: names_ages.ads

```

1 package Names_Ages is
2
3   Max_People : constant Positive := 10;
4
5   subtype Name_Type is String (1 .. 50);
6
7   type Age_Type is new Natural;
8
9   type Person is record
10     Name : Name_Type;
11     Age  : Age_Type;
12   end record;
13
14   -- Add type declaration for People_Array record:
15   --
16   -- type People_Array is ...;
17
18   -- Replace type declaration for People record. You may use the
19   -- following template:
20   --
21   -- type People is record
22   --   People_A : People_Array ...;
23   --   Last_Valid : Natural;
24   -- end record;
25   --
26   type People is null record;
27
28   procedure Reset (P : in out People);
29
30   procedure Add (P : in out People;
31                 Name : String);
32
33   function Get (P : People;
34                Name : String) return Age_Type;
35
36   procedure Update (P : in out People;
37                    Name : String;
38                    Age : Age_Type);
39
40   procedure Display (P : People);

```

(continues on next page)

(continued from previous page)

```
41
42 end Names_Ages;
```

Listing 2559: names_ages.adb

```
1  with Ada.Text_IO;          use Ada.Text_IO;
2  with Ada.Strings;          use Ada.Strings;
3  with Ada.Strings.Fixed;    use Ada.Strings.Fixed;
4
5  package body Names_Ages is
6
7      procedure Reset (P : in out People) is
8      begin
9          null;
10     end Reset;
11
12     procedure Add (P : in out People;
13                   Name : String) is
14     begin
15         null;
16     end Add;
17
18     function Get (P : People;
19                  Name : String) return Age_Type is
20     begin
21         return 0;
22     end Get;
23
24     procedure Update (P : in out People;
25                       Name : String;
26                       Age : Age_Type) is
27     begin
28         null;
29     end Update;
30
31     procedure Display (P : People) is
32     begin
33         null;
34     end Display;
35
36 end Names_Ages;
```

Listing 2560: main.adb

```
1  with Ada.Command_Line;    use Ada.Command_Line;
2  with Ada.Text_IO;         use Ada.Text_IO;
3
4  with Names_Ages;          use Names_Ages;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Names_Ages_Chk,
9           Get_Age_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     P : People;
13     begin
14         case TC is
15         when Names_Ages_Chk =>
16             Reset (P);
```

(continues on next page)

(continued from previous page)

```

17      Add (P, "John");
18      Add (P, "Patricia");
19      Add (P, "Josh");
20      Display (P);
21      Update (P, "John", 18);
22      Update (P, "Patricia", 35);
23      Update (P, "Josh", 53);
24      Display (P);
25      when Get_Age_Chk =>
26          Reset (P);
27          Add (P, "Peter");
28          Update (P, "Peter", 45);
29          Put_Line ("Peter is "
30                  & Age_Type'Image (Get (P, "Peter"))
31                  & " years old.");
32      end case;
33  end Check;
34
35  begin
36      if Argument_Count < 1 then
37          Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
38          return;
39      elsif Argument_Count > 1 then
40          Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
41      end if;
42
43      Check (Test_Case_Index'Value (Argument (1)));
44  end Main;

```


MORE ABOUT TYPES

123.1 Aggregate Initialization

Goal: initialize records and arrays using aggregates.

Steps:

1. Implement the Aggregates package.
 1. Create the record type Rec.
 2. Create the array type Int_Arr.
 3. Implement the Init procedure that outputs a record of Rec type.
 4. Implement the Init_Some procedure.
 5. Implement the Init procedure that outputs an array of Int_Arr type.

Requirements:

1. Record type Rec has four components of **Integer** type. These are the components with the corresponding default values:
 - W = 10
 - X = 11
 - Y = 12
 - Z = 13
2. Array type Int_Arr has 20 elements of **Integer** type (with indices ranging from 1 to 20).
3. The first Init procedure outputs a record of Rec type where:
 1. X is initialized with 100,
 2. Y is initialized with 200, and
 3. the remaining elements use their default values.
4. Procedure Init_Some outputs an array of Int_Arr type where:
 1. the first five elements are initialized with the value 99, and
 2. the remaining elements are initialized with the value 100.
5. The second Init procedure outputs an array of Int_Arr type where:
 1. all elements are initialized with the value 5.

Listing 2561: aggregates.ads

```

1 package Aggregates is
2
3     -- type Rec is ...;
4
5     -- type Int_Arr is ...;
6
7     procedure Init;
8
9     -- procedure Init_Some ...;
10
11    -- procedure Init ...;
12
13 end Aggregates;

```

Listing 2562: aggregates.adb

```

1 package body Aggregates is
2
3     procedure Init is null;
4
5 end Aggregates;

```

Listing 2563: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Aggregates;       use Aggregates;
5
6 procedure Main is
7     -- Remark: the following line is not relevant.
8     F : array (1 .. 10) of Float := (others => 42.42)
9     with Unreferenced;
10
11    type Test_Case_Index is
12        (Default_Rec_Chk,
13         Init_Rec_Chk,
14         Init_Some_Arr_Chk,
15         Init_Arr_Chk);
16
17    procedure Check (TC : Test_Case_Index) is
18        A : Int_Arr;
19        R : Rec;
20        DR : constant Rec := (others => <>);
21    begin
22        case TC is
23            when Default_Rec_Chk =>
24                R := DR;
25                Put_Line ("Record Default:");
26                Put_Line ("W => " & Integer'Image (R.W));
27                Put_Line ("X => " & Integer'Image (R.X));
28                Put_Line ("Y => " & Integer'Image (R.Y));
29                Put_Line ("Z => " & Integer'Image (R.Z));
30            when Init_Rec_Chk =>
31                Init (R);
32                Put_Line ("Record Init:");
33                Put_Line ("W => " & Integer'Image (R.W));
34                Put_Line ("X => " & Integer'Image (R.X));
35                Put_Line ("Y => " & Integer'Image (R.Y));
36                Put_Line ("Z => " & Integer'Image (R.Z));

```

(continues on next page)

(continued from previous page)

```

37     when Init_Some_Arr_Chk =>
38         Init_Some (A);
39         Put_Line ("Array Init_Some:");
40         for I in A'Range loop
41             Put_Line (Integer'Image (I) & " "
42                     & Integer'Image (A (I)));
43         end loop;
44     when Init_Arr_Chk =>
45         Init (A);
46         Put_Line ("Array Init:");
47         for I in A'Range loop
48             Put_Line (Integer'Image (I) & " "
49                     & Integer'Image (A (I)));
50         end loop;
51     end case;
52 end Check;
53
54 begin
55     if Argument_Count < 1 then
56         Put_Line ("ERROR: missing arguments! Exiting...");
57         return;
58     elsif Argument_Count > 1 then
59         Put_Line ("Ignoring additional arguments...");
60     end if;
61
62     Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

123.2 Versioning

Goal: implement a simple package for source-code versioning.

Steps:

1. Implement the Versioning package.
 1. Declare the record type Version.
 2. Implement the Convert function that returns a string.
 3. Implement the Convert function that returns a floating-point number.

Requirements:

1. Record type Version has the following components of **Natural** type:
 1. Major,
 2. Minor, and
 3. Maintenance.
2. The first Convert function returns a string containing the version number.
3. The second Convert function returns a floating-point value.
 1. For this floating-point value:
 1. the number before the decimal point must correspond to the major number, and
 2. the number after the decimal point must correspond to the minor number.
 3. the maintenance number is ignored.
 2. For example, version "1.3.5" is converted to the floating-point value 1.3.

3. An obvious limitation of this function is that it can only handle one-digit numbers for the minor component.

- For example, we cannot convert version "1.10.0" to a reasonable value with the approach described above. The result of the call `Convert ((1, 10, 0))` is therefore unspecified.
- For the scope of this exercise, only version numbers with one-digit components are checked.

Remarks:

1. We use overloading for the `Convert` functions.
2. For the function `Convert` that returns a string, you can make use of the `Image_Trim` function, as indicated in the source-code below — see package body of `Versioning`.

Listing 2564: `versioning.ads`

```
1 package Versioning is
2
3   -- type Version is record...
4
5   -- function Convert ...
6
7   -- function Convert
8
9 end Versioning;
```

Listing 2565: `versioning.adb`

```
1 with Ada.Strings; use Ada.Strings;
2 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4 package body Versioning is
5
6   function Image_Trim (N : Natural) return String is
7     S_N : constant String := Trim (Natural'Image (N), Left);
8   begin
9     return S_N;
10  end Image_Trim;
11
12   -- function Convert ...
13   --   S_Major : constant String := Image_Trim (V.Major);
14   --   S_Minor : constant String := Image_Trim (V.Minor);
15   --   S_Maint : constant String := Image_Trim (V.Maintenance);
16   -- begin
17   -- end Convert;
18
19   -- function Convert ...
20   -- begin
21   -- end Convert;
22
23 end Versioning;
```

Listing 2566: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Versioning; use Versioning;
5
6 procedure Main is
```

(continues on next page)

(continued from previous page)

```

7  type Test_Case_Index is
8      (Ver_String_Chk,
9       Ver_Float_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12      V : constant Version := (1, 3, 23);
13  begin
14      case TC is
15          when Ver_String_Chk =>
16              Put_Line (Convert (V));
17          when Ver_Float_Chk =>
18              Put_Line (Float'Image (Convert (V)));
19      end case;
20  end Check;
21
22  begin
23      if Argument_Count < 1 then
24          Put_Line ("ERROR: missing arguments! Exiting...");
25          return;
26      elsif Argument_Count > 1 then
27          Put_Line ("Ignoring additional arguments...");
28      end if;
29
30      Check (Test_Case_Index'Value (Argument (1)));
31  end Main;

```

123.3 Simple todo list

Goal: implement a simple to-do list system.

Steps:

1. Implement the `Todo_Lists` package.
 1. Declare the `Todo_Item` type.
 2. Declare the `Todo_List` type.
 3. Implement the `Add` procedure.
 4. Implement the `Display` procedure.

Requirements:

1. `Todo_Item` type is used to store a to-do item.
 1. It should be implemented as an access type to strings.
2. `Todo_Items` type is an array of to-do items.
 1. It should be implemented as an unconstrained array with positive range.
3. `Todo_List` type is the container for all to-do items.
 1. This record type must have a discriminant for the maximum number of elements of the list.
 2. In order to store the to-do items, it must contain a component named `Items` of `Todo_Items` type.
 3. Don't forget to keep track of the last element added to the list!
 - You should declare a `Last` component in the record.
4. Procedure `Add` adds items (of `Todo_Item` type) to the list (of `Todo_List` type).

1. This requires allocating a string for the access type.
2. An item can only be added to the list if the list isn't full yet — see next point for details on error handling.
5. Since the number of items that can be stored on the list is limited, the list might eventually become full in a call to Add.
 1. You must write code in the implementation of the Add procedure that verifies this condition.
 2. If the procedure detects that the list is full, it must display the following message: "ERROR: list is full!".
6. Procedure Display is used to display all to-do items.
 1. The header (first line) must be T0-**DO** LIST.
 2. It must display one item per line.

Remarks:

1. We use access types and unconstrained arrays in the implementation of the Todo_Lists package.

Listing 2567: todo_lists.ads

```
1 package Todo_Lists is
2
3   -- Replace by actual type declaration
4   type Todo_Item is null record;
5
6   -- Replace by actual type declaration
7   type Todo_Items is null record;
8
9   -- Replace by actual type declaration
10  type Todo_List is null record;
11
12  procedure Add (Todos : in out Todo_List;
13               Item : String);
14
15  procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;
```

Listing 2568: todo_lists.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5   procedure Add (Todos : in out Todo_List;
6               Item : String) is
7   begin
8     Put_Line ("ERROR: list is full!");
9   end Add;
10
11  procedure Display (Todos : Todo_List) is
12  begin
13    null;
14  end Display;
15
16 end Todo_Lists;
```

Listing 2569: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;        use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List (10);
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27         end case;
28     end Check;
29
30 begin
31     if Argument_Count < 1 then
32         Put_Line ("ERROR: missing arguments! Exiting...");
33         return;
34     elsif Argument_Count > 1 then
35         Put_Line ("Ignoring additional arguments...");
36     end if;
37
38     Check (Test_Case_Index'Value (Argument (1)));
39 end Main;

```

123.4 Price list

Goal: implement a list containing prices

Steps:

1. Implement the Price_Lists package.
 1. Declare the Price_Type type.
 2. Declare the Price_List record.
 3. Implement the Reset procedure.
 4. Implement the Add procedure.
 5. Implement the Get function.
 6. Implement the Display procedure.

Requirements:

1. Price_Type is a decimal fixed-point data type with a delta of two digits (e.g. 0.01) and twelve digits in total.
2. Price_List is a record type that contains the price list.
 1. This record type must have a discriminant for the maximum number of elements of the list.
3. Procedure Reset resets the list.
4. Procedure Add adds a price to the list.
 1. You should keep track of the last element added to the list.
5. Function Get retrieves a price from the list using an index.
 1. This function returns a record instance of Price_Result type.
 2. Price_Result is a variant record containing:
 1. the Boolean component Ok, and
 2. the component Price (of Price_Type).
 3. The returned value of Price_Result type is one of the following:
 1. If the index specified in a call to Get contains a valid (initialized) price, then
 - Ok is set to **True**, and
 - the Price component contains the price for that index.
 2. Otherwise:
 - Ok is set to **False**, and
 - the Price component is not available.
6. Procedure Display shows all prices from the list.
 1. The header (first line) must be PRICE LIST.
 2. The remaining lines contain one price per line.
 3. For example:
 - For the following code:

```
procedure Test is
  L : Price_List (10);
begin
  Reset (L);
  Add (L, 1.45);
  Add (L, 2.37);
  Display (L);
end Test;
```

- The output is:

```
PRICE LIST
1.45
2.37
```

Remarks:

1. To implement the package, you'll use the following features of the Ada language:
 1. decimal fixed-point types;
 2. records with discriminants;
 3. dynamically-sized record types;

4. variant records.
2. For record type `Price_List`, you may use an unconstrained array as a component of the record and use the discriminant in the component declaration.

Listing 2570: price_lists.ads

```

1 package Price_Lists is
2
3   -- Replace by actual type declaration
4   type Price_Type is new Float;
5
6   -- Replace by actual type declaration
7   type Price_List is null record;
8
9   -- Replace by actual type declaration
10  type Price_Result is null record;
11
12  procedure Reset (Prices : in out Price_List);
13
14  procedure Add (Prices : in out Price_List;
15               Item   : Price_Type);
16
17  function Get (Prices : Price_List;
18              Idx     : Positive) return Price_Result;
19
20  procedure Display (Prices : Price_List);
21
22 end Price_Lists;

```

Listing 2571: price_lists.adb

```

1 package body Price_Lists is
2
3   procedure Reset (Prices : in out Price_List) is
4   begin
5     null;
6   end Reset;
7
8   procedure Add (Prices : in out Price_List;
9                Item   : Price_Type) is
10  begin
11    null;
12  end Add;
13
14  function Get (Prices : Price_List;
15              Idx     : Positive) return Price_Result is
16  begin
17    null;
18  end Get;
19
20  procedure Display (Prices : Price_List) is
21  begin
22    null;
23  end Display;
24
25 end Price_Lists;

```

Listing 2572: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```

3
4 with Price_Lists;          use Price_Lists;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Price_Type_Chk,
9      Price_List_Chk,
10     Price_List_Get_Chk);
11
12   procedure Check (TC : Test_Case_Index) is
13     L : Price_List (10);
14
15     procedure Local_Init_List is
16     begin
17       Reset (L);
18       Add (L, 1.45);
19       Add (L, 2.37);
20       Add (L, 3.21);
21       Add (L, 4.14);
22       Add (L, 5.22);
23       Add (L, 6.69);
24       Add (L, 7.77);
25       Add (L, 8.14);
26       Add (L, 9.99);
27       Add (L, 10.01);
28     end Local_Init_List;
29
30     procedure Get_Display (Idx : Positive) is
31       R : constant Price_Result := Get (L, Idx);
32     begin
33       Put_Line ("Attempt Get # " & Positive'Image (Idx));
34       if R.Ok then
35         Put_Line ("Element # " & Positive'Image (Idx)
36           & " => " & Price_Type'Image (R.Price));
37       else
38         declare
39         begin
40           Put_Line ("Element # " & Positive'Image (Idx)
41             & " => " & Price_Type'Image (R.Price));
42         exception
43           when others =>
44             Put_Line ("Element not available (as expected)");
45         end;
46       end if;
47     end Get_Display;
48
49   begin
50     case TC is
51     when Price_Type_Chk =>
52       Put_Line ("The delta value of Price_Type is "
53         & Price_Type'Image (Price_Type'Delta) & ";");
54       Put_Line ("The minimum value of Price_Type is "
55         & Price_Type'Image (Price_Type'First) & ";");
56       Put_Line ("The maximum value of Price_Type is "
57         & Price_Type'Image (Price_Type'Last) & ";");
58     when Price_List_Chk =>
59       Local_Init_List;
60       Display (L);
61     when Price_List_Get_Chk =>
62       Local_Init_List;
63

```

(continues on next page)

(continued from previous page)

```
64         Get_Display (5);
65         Get_Display (40);
66     end case;
67 end Check;
68
69 begin
70     if Argument_Count < 1 then
71         Put_Line ("ERROR: missing arguments! Exiting...");
72         return;
73     elsif Argument_Count > 1 then
74         Put_Line ("Ignoring additional arguments...");
75     end if;
76
77     Check (Test_Case_Index'Value (Argument (1)));
78 end Main;
```


PRIVACY

124.1 Directions

Goal: create a package that handles directions and geometric angles using a previous implementation.

Steps:

1. Fix the implementation of the `Test_Directions` procedure.

Requirements:

1. The implementation of the `Test_Directions` procedure must compile correctly.

Remarks:

1. This exercise is based on the *Directions* exercise from the [Records](#) (page 2185) labs.
 1. In this version, however, `Ext_Angle` is a private type.
2. In the implementation of the `Test_Directions` procedure below, the Ada developer tried to initialize `All_Directions` — an array of `Ext_Angle` type — with aggregates.
 1. Since we now have a private type, the compiler complains about this initialization.
3. To fix the implementation of the `Test_Directions` procedure, you should use the appropriate function from the `Directions` package.
4. The initialization of `All_Directions` in the code below contains a consistency error where the angle doesn't match the assessed direction.
 1. See if you can spot this error!
 2. This kind of errors can happen when record components that have correlated information are initialized individually without consistency checks — using private types helps to avoid the problem by requiring initialization routines that can enforce consistency.

Listing 2573: `directions.ads`

```
1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northwest,
8      West,
9      Southwest,
10     South,
11     Southeast,
12     East);
13
```

(continues on next page)

(continued from previous page)

```

14  function To_Direction (N : Angle_Mod) return Direction;
15
16  type Ext_Angle is private;
17
18  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
19
20  procedure Display (N : Ext_Angle);
21
22 private
23
24  type Ext_Angle is record
25      Angle_Elem      : Angle_Mod;
26      Direction_Elem : Direction;
27  end record;
28
29  end Directions;

```

Listing 2574: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0      => return East;
18              when 1 .. 89 => return Northwest;
19              when 90     => return North;
20              when 91 .. 179 => return Northwest;
21              when 180    => return West;
22              when 181 .. 269 => return Southwest;
23              when 270    => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem      => N,
31                 Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34  end Directions;

```

Listing 2575: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
4      type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;

```

(continues on next page)

(continued from previous page)

```

5
6  All_Directions : constant Ext_Angle_Array (1 .. 6)
7      := ((0,   East),
8          (45,  Northwest),
9          (90,  North),
10         (91,  North),
11         (180, West),
12         (270, South));
13
14 begin
15     for I in All_Directions'Range loop
16         Display (All_Directions (I));
17     end loop;
18
19 end Test_Directions;

```

Listing 2576: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Direction_Chk =>
14                 Test_Directions;
15         end case;
16     end Check;
17
18     begin
19         if Argument_Count < 1 then
20             Put_Line ("ERROR: missing arguments! Exiting...");
21             return;
22         elsif Argument_Count > 1 then
23             Put_Line ("Ignoring additional arguments...");
24         end if;
25
26         Check (Test_Case_Index'Value (Argument (1)));
27     end Main;

```

124.2 Limited Strings

Goal: work with **limited private** types.

Steps:

1. Implement the Limited_Strings package.
 1. Implement the Copy function.
 2. Implement the = operator.

Requirements:

1. For both Copy and =, the two parameters may refer to strings with different lengths.

We'll limit the implementation to just take the minimum length:

1. In case of copying the string "Hello World" to a string with 5 characters, the copied string is "Hello":

```
S1 : constant Lim_String := Init ("Hello World");
S2 :          Lim_String := Init (5);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello".
```

2. When comparing "Hello World" to "Hello", the = operator indicates that these strings are equivalent:

```
S1 : constant Lim_String := Init ("Hello World");
S2 : constant Lim_String := Init ("Hello");
begin
  if S1 = S2 then
    -- True => This branch gets selected.
```

2. When copying from a short string to a longer string, the remaining characters of the longer string must be initialized with underscores (_). For example:

```
S1 : constant Lim_String := Init ("Hello");
S2 :          Lim_String := Init (10);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello_____".
```

Remarks:

1. As we've discussed in the course:
 1. Variables of limited types have the following limitations:
 - they cannot be assigned to;
 - they don't have an equality operator (=).
 2. We can, however, define our own, custom subprograms to circumvent these limitations:
 - In order to copy instances of a limited type, we can define a custom Copy procedure.
 - In order to compare instances of a limited type, we can define an = operator.
2. You can use the `Min_Last` constant — which is already declared in the implementation of these subprograms — in the code you write.
3. Some details about the `Limited_Strings` package:
 1. The `Lim_String` type acts as a container for strings.
 1. In the the private part, `Lim_String` is declared as an access type to a **String**.
 2. There are two versions of the `Init` function that initializes an object of `Lim_String` type:
 1. The first one takes another string.
 2. The second one receives the number of characters for a string *container*.
 3. Procedure `Put_Line` displays object of `Lim_String` type.
 4. The design and implementation of the `Limited_Strings` package is very simplistic.
 1. A good design would have better handling of access types, for example.

Listing 2577: limited_strings.ads

```

1 package Limited_Strings is
2
3     type Lim_String is limited private;
4
5     function Init (S : String) return Lim_String;
6
7     function Init (Max : Positive) return Lim_String;
8
9     procedure Put_Line (LS : Lim_String);
10
11    procedure Copy (From : Lim_String;
12                  To   : in out Lim_String);
13
14    function "=" (Ref, Dut : Lim_String) return Boolean;
15
16 private
17
18     type Lim_String is access String;
19
20 end Limited_Strings;
```

Listing 2578: limited_strings.adb

```

1 with Ada.Text_IO;
2
3 package body Limited_Strings
4 is
5
6     function Init (S : String) return Lim_String is
7         LS : constant Lim_String := new String'(S);
8     begin
9         return Ls;
10    end Init;
11
12    function Init (Max : Positive) return Lim_String is
13        LS : constant Lim_String := new String (1 .. Max);
14    begin
15        LS.all := (others => '_');
16        return LS;
17    end Init;
18
19    procedure Put_Line (LS : Lim_String) is
20    begin
21        Ada.Text_IO.Put_Line (LS.all);
22    end Put_Line;
23
24    function Get_Min_Last (A, B : Lim_String) return Positive is
25    begin
26        return Positive'Min (A'Last, B'Last);
27    end Get_Min_Last;
28
29    procedure Copy (From : Lim_String;
30                  To   : in out Lim_String) is
31        Min_Last : constant Positive := Get_Min_Last (From, To);
32    begin
33        -- Complete the implementation!
34        null;
35    end;
```

(continues on next page)

(continued from previous page)

```
37 function "=" (Ref, Dut : Lim_String) return Boolean is
38   Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39 begin
40   -- Complete the implementation!
41   return True;
42 end;
43
44 end Limited_Strings;
```

Listing 2579: check_lim_string.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Limited_Strings; use Limited_Strings;
4
5 procedure Check_Lim_String is
6   S : constant String := "-----";
7   S1 : constant Lim_String := Init ("Hello World");
8   S2 : constant Lim_String := Init (30);
9   S3 : Lim_String := Init (5);
10  S4 : Lim_String := Init (S & S & S);
11 begin
12   Put ("S1 => ");
13   Put_Line (S1);
14   Put ("S2 => ");
15   Put_Line (S2);
16
17   if S1 = S2 then
18     Put_Line ("S1 is equal to S2.");
19   else
20     Put_Line ("S1 isn't equal to S2.");
21   end if;
22
23   Copy (From => S1, To => S3);
24   Put ("S3 => ");
25   Put_Line (S3);
26
27   if S1 = S3 then
28     Put_Line ("S1 is equal to S3.");
29   else
30     Put_Line ("S1 isn't equal to S3.");
31   end if;
32
33   Copy (From => S1, To => S4);
34   Put ("S4 => ");
35   Put_Line (S4);
36
37   if S1 = S4 then
38     Put_Line ("S1 is equal to S4.");
39   else
40     Put_Line ("S1 isn't equal to S4.");
41   end if;
42 end Check_Lim_String;
```

Listing 2580: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Check_Lim_String;
```

(continues on next page)

(continued from previous page)

```

5
6 procedure Main is
7   type Test_Case_Index is
8     (Lim_String_Chk);
9
10  procedure Check (TC : Test_Case_Index) is
11  begin
12    case TC is
13      when Lim_String_Chk =>
14        Check_Lim_String;
15    end case;
16  end Check;
17
18 begin
19   if Argument_Count < 1 then
20     Put_Line ("ERROR: missing arguments! Exiting...");
21     return;
22   elsif Argument_Count > 1 then
23     Put_Line ("Ignoring additional arguments...");
24   end if;
25
26   Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

124.3 Bonus exercise

In previous labs, we had many source-code snippets containing records that could be declared private. The source-code for the exercise above (*Directions*) is an example: we've modified the type declaration of `Ext_Angle`, so that the record is now private. Encapsulating the record components — by declaring record components in the private part — makes the code safer. Also, because many of the code snippets weren't making use of record components directly (but handling record types via the API instead), they continue to work fine after these modifications.

This exercise doesn't contain any source-code. In fact, the **goal** here is to modify previous labs, so that the record declarations are made private. You can look into those labs, modify the type declarations, and recompile the code. The corresponding test-cases must still pass.

If no other changes are needed apart from changes in the declaration, then that indicates we have used good programming techniques in the original code. On the other hand, if further changes are needed, then you should investigate why this is the case.

Also note that, in some cases, you can move support types into the private part of the specification without affecting its compilation. This is the case, for example, for the `People_Array` type of the *List of Names* lab mentioned below. You should, in fact, keep only relevant types and subprograms in the public part and move all support declarations to the private part of the specification whenever possible.

Below, you find the selected labs that you can work on, including changes that you should make. In case you don't have a working version of the source-code of previous labs, you can look into the corresponding solutions.

124.3.1 Colors

Chapter: *Records* (page 2185)

Steps:

1. Change declaration of RGB type to **private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

124.3.2 List of Names

Chapter: *Arrays* (page 2195)

Steps:

1. Change declaration of Person and People types to **limited private**.
2. Move type declaration of People_Array to private part.

Requirements:

1. Implementation must compile correctly and test cases must pass.

124.3.3 Price List

Chapter: *More About Types* (page 2213)

Steps:

1. Change declaration of Price_List type to **limited private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

GENERICCS

125.1 Display Array

Goal: create a generic procedure that displays the elements of an array.

Steps:

1. Implement the generic procedure `Display_Array`.

Requirements:

1. Generic procedure `Display_Array` displays the elements of an array.
 1. It uses the following scheme:
 - First, it displays a header.
 - Then, it displays the elements of the array.
 2. When displaying the elements, it must:
 - use one line per element, and
 - include the corresponding index of the array.
 3. This is the expected format:

```
<HEADER>
<index #1>: <element #1>
<index #2>: <element #2>
...
```

4. For example:

- For the following code:

```
procedure Test is
  A : Int_Array (1 .. 2) := (1, 5);
begin
  Display_Int_Array ("Elements of A", A);;
end Test;
```

- The output is:

```
Elements of A
1: 1
2: 5
```

2. These are the formal parameters of the procedure:
 1. a range type `T_Range` for the the array;
 2. a formal type `T_Element` for the elements of the array;

- This type must be declared in such a way that it can be mapped to any type in the instantiation — including record types.
3. an array type `T_Array` using the `T_Range` and `T_Element` types;
 4. a function `Image` that converts a variable of `T_Element` type to a **String**.

Listing 2581: `display_array.ads`

```
1 generic
2 procedure Display_Array (Header : String;
3                        A       : T_Array);
```

Listing 2582: `display_array.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Array (Header : String;
4                        A       : T_Array) is
5 begin
6     null;
7 end Display_Array;
```

Listing 2583: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Array;
5
6 procedure Main is
7     type Test_Case_Index is (Int_Array_Chk,
8                             Point_Array_Chk);
9
10    procedure Test_Int_Array is
11        type Int_Array is array (Positive range <>) of Integer;
12
13        procedure Display_Int_Array is new
14            Display_Array (T_Range => Positive,
15                          T_Element => Integer,
16                          T_Array  => Int_Array,
17                          Image    => Integer'Image);
18
19        A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20    begin
21        Display_Int_Array ("Integers", A);
22    end Test_Int_Array;
23
24    procedure Test_Point_Array is
25        type Point is record
26            X : Float;
27            Y : Float;
28        end record;
29
30        type Point_Array is array (Natural range <>) of Point;
31
32        function Image (P : Point) return String is
33        begin
34            return "(" & Float'Image (P.X)
35                & ", " & Float'Image (P.Y) & ")";
36        end Image;
37
```

(continues on next page)

(continued from previous page)

```

38     procedure Display_Point_Array is new
39         Display_Array (T_Range => Natural,
40                       T_Element => Point,
41                       T_Array => Point_Array,
42                       Image => Image);
43
44     A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                           (5.0, 2.0), (-0.5, 2.0));
46
47     begin
48         Display_Point_Array ("Points", A);
49     end Test_Point_Array;
50
51     procedure Check (TC : Test_Case_Index) is
52     begin
53         case TC is
54             when Int_Array_Chk =>
55                 Test_Int_Array;
56             when Point_Array_Chk =>
57                 Test_Point_Array;
58         end case;
59     end Check;
60
61     begin
62         if Argument_Count < 1 then
63             Put_Line ("ERROR: missing arguments! Exiting...");
64             return;
65         elsif Argument_Count > 1 then
66             Put_Line ("Ignoring additional arguments...");
67         end if;
68
69         Check (Test_Case_Index'Value (Argument (1)));
70     end Main;

```

125.2 Average of Array of Float

Goal: create a generic function that calculates the average of an array of floating-point elements.

Steps:

1. Declare and implement the generic function Average.

Requirements:

1. Generic function Average calculates the average of an array containing floating-point values of arbitrary precision.
2. Generic function Average must contain the following formal parameters:
 1. a range type T_Range for the array;
 2. a formal type T_Element that can be mapped to floating-point types of arbitrary precision;
 3. an array type T_Array using T_Range and T_Element;

Remarks:

1. You should use the **Float** type for the accumulator.

Listing 2584: average.ads

```

1 generic
2 function Average (A : T_Array) return T_Element;
```

Listing 2585: average.adb

```

1 function Average (A : T_Array) return T_Element is
2 begin
3     return 0.0;
4 end Average;
```

Listing 2586: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Average;
5
6 procedure Main is
7     type Test_Case_Index is (Float_Array_Chk,
8                               Digits_7_Float_Array_Chk);
9
10    procedure Test_Float_Array is
11        type Float_Array is array (Positive range <>) of Float;
12
13        function Average_Float is new
14            Average (T_Range => Positive,
15                    T_Element => Float,
16                    T_Array => Float_Array);
17
18        A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19    begin
20        Put_Line ("Average: " & Float'Image (Average_Float (A)));
21    end Test_Float_Array;
22
23    procedure Test_Digits_7_Float_Array is
24        type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26        type Float_Array is
27            array (Integer range <>) of Custom_Float;
28
29        function Average_Float is new
30            Average (T_Range => Integer,
31                    T_Element => Custom_Float,
32                    T_Array => Float_Array);
33
34        A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
35    begin
36        Put_Line ("Average: "
37                & Custom_Float'Image (Average_Float (A)));
38    end Test_Digits_7_Float_Array;
39
40    procedure Check (TC : Test_Case_Index) is
41    begin
42        case TC is
43            when Float_Array_Chk =>
44                Test_Float_Array;
45            when Digits_7_Float_Array_Chk =>
46                Test_Digits_7_Float_Array;
47        end case;
```

(continues on next page)

(continued from previous page)

```

48     end Check;
49
50 begin
51     if Argument_Count < 1 then
52         Put_Line ("ERROR: missing arguments! Exiting...");
53         return;
54     elsif Argument_Count > 1 then
55         Put_Line ("Ignoring additional arguments...");
56     end if;
57
58     Check (Test_Case_Index'Value (Argument (1)));
59 end Main;

```

125.3 Average of Array of Any Type

Goal: create a generic function that calculates the average of an array of elements of any arbitrary type.

Steps:

1. Declare and implement the generic function Average.
2. Implement the test procedure Test_Item.
 1. Declare the F_IO package.
 2. Implement the Get_Total function for the Item type.
 3. Implement the Get_Price function for the Item type.
 4. Declare the Average_Total function.
 5. Declare the Average_Price function.

Requirements:

1. Generic function Average calculates the average of an array containing elements of any arbitrary type.
2. Generic function Average has the same formal parameters as in the previous exercise, except for:
 1. T_Element, which is now a formal type that can be mapped to any arbitrary type.
 2. To_Float, which is an *additional* formal parameter.
 - To_Float is a function that converts the arbitrary element of T_Element type to the **Float** type.
3. Procedure Test_Item is used to test the generic Average procedure for a record type (Item).
 1. Record type Item contains the Quantity and Price components.
4. The following functions have to be implemented to be used for the formal To_Float function parameter:
 1. For the Decimal type, the function is pretty straightforward: it simply returns the floating-point value converted from the decimal type.
 2. For the Item type, two functions must be created to convert to floating-point type:
 1. Get_Total, which returns the multiplication of the quantity and the price components of the Item type;
 2. Get_Price, which returns just the price.

5. The generic function `Average` must be instantiated as follows:

1. For the `Item` type, you must:

1. declare the `Average_Total` function (as an instance of `Average`) using the `Get_Total` for the `To_Float` parameter;
2. declare the `Average_Price` function (as an instance of `Average`) using the `Get_Price` for the `To_Float` parameter.

6. You must use the `Put` procedure from `Ada.Text_IO.Float_IO`.

1. The generic standard package `Ada.Text_IO.Float_IO` must be instantiated as `F_IO` in the test procedures.
2. This is the specification of the `Put` procedure, as described in the appendix A.10.9 of the Ada Reference Manual:

```
procedure Put(Item : in Num;
             Fore : in Field := Default_Fore;
             Aft  : in Field := Default_Aft;
             Exp  : in Field := Default_Exp);
```

3. This is the expected format when calling `Put` from `Float_IO`:

Function	Fore	Aft	Exp
Test_Item	3	2	0

Remarks:

1. In this exercise, you'll abstract the `Average` function from the previous exercises a step further.
 1. In this case, the function shall be able to calculate the average of any arbitrary type — including arrays containing elements of record types.
 2. Since record types can be composed by many components of different types, we need to provide a way to indicate which component (or components) of the record will be used when calculating the average of the array.
 3. This problem is solved by specifying a `To_Float` function as a formal parameter, which converts the arbitrary element of `T_Element` type to the **Float** type.
 4. In the implementation of the `Average` function, we use the `To_Float` function and calculate the average using a floating-point variable.

Listing 2587: `average.ads`

```
1 generic
2 function Average (A : T_Array) return Float;
```

Listing 2588: `average.adb`

```
1 function Average (A : T_Array) return Float is
2 begin
3   null;
4 end Average;
```

Listing 2589: `test_item.ads`

```
1 procedure Test_Item;
```

Listing 2590: test_item.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Average;
4
5  procedure Test_Item is
6      type Amount is delta 0.01 digits 12;
7
8      type Item is record
9          Quantity : Natural;
10         Price    : Amount;
11     end record;
12
13     type Item_Array is
14         array (Positive range <>) of Item;
15
16     A : constant Item_Array (1 .. 4)
17         := ((Quantity => 5,   Price => 10.00),
18            (Quantity => 80,  Price => 2.50),
19            (Quantity => 40,  Price => 5.00),
20            (Quantity => 20,  Price => 12.50));
21
22     begin
23         Put ("Average per item & quantity: ");
24         F_IO.Put (Average_Total (A));
25         New_Line;
26
27         Put ("Average price:                ");
28         F_IO.Put (Average_Price (A));
29         New_Line;
30     end Test_Item;

```

Listing 2591: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Item;
5
6  procedure Main is
7      type Test_Case_Index is (Item_Array_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10     begin
11         case TC is
12             when Item_Array_Chk =>
13                 Test_Item;
14         end case;
15     end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;

```

125.4 Generic list

Goal: create a system based on a generic list to add and displays elements.

Steps:

1. Declare and implement the generic package `Gen_List`.
 1. Implement the `Init` procedure.
 2. Implement the `Add` procedure.
 3. Implement the `Display` procedure.

Requirements:

1. Generic package `Gen_List` must have the following subprograms:
 1. Procedure `Init` initializes the list.
 2. Procedure `Add` adds an item to the list.
 1. This procedure must contain a `Status` output parameter that is set to **False** when the list was full — i.e. if the procedure failed while trying to add the item;
 3. Procedure `Display` displays the complete list.
 1. This includes the *name* of the list and its elements — using one line per element.
 2. This is the expected format:

```
<NAME>
<element #1>
<element #2>
...
```

2. Generic package `Gen_List` has these formal parameters:
 1. an arbitrary formal type `Item`;
 2. an unconstrained array type `Items` of `Item` element with positive range;
 3. the `Name` parameter containing the name of the list;
 - This must be a formal input object of **String** type.
 - It must be used in the `Display` procedure.
 4. an actual array `List_Array` to store the list;
 - This must be a formal **in out** object of `Items` type.
 5. the variable `Last` to store the index of the last element;
 - This must be a formal **in out** object of **Natural** type.
 6. a procedure `Put` for the `Item` type.
 - This procedure is used in the `Display` procedure to display individual elements of the list.
3. The test procedure `Test_Int` is used to test a list of elements of **Integer** type.
4. For both test procedures, you must:
 1. add missing type declarations;
 2. declare and implement a `Put` procedure for individual elements of the list;
 3. declare instances of the `Gen_List` package.
 - For the `Test_Int` procedure, declare the `Int_List` package.

Remarks:

1. In previous labs, you've been implementing lists for a variety of types.
 - The *List of Names* exercise from the [Arrays](#) (page 2195) labs is an example.
 - In this exercise, you have to abstract those implementations to create the generic Gen_List package.

Listing 2592: gen_list.ads

```

1 generic
2 package Gen_List is
3
4     procedure Init;
5
6     procedure Add (I      : Item;
7                   Status : out Boolean);
8
9     procedure Display;
10
11 end Gen_List;
```

Listing 2593: gen_list.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_List is
4
5     procedure Init is
6     begin
7         null;
8     end Init;
9
10    procedure Add (I      : Item;
11                  Status : out Boolean) is
12    begin
13        null;
14    end Add;
15
16    procedure Display is
17    begin
18        null;
19    end Display;
20
21 end Gen_List;
```

Listing 2594: test_int.ads

```

1 procedure Test_Int;
```

Listing 2595: test_int.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_List;
4
5 procedure Test_Int is
6
7     type Integer_Array is array (Positive range <>) of Integer;
8
9     A : Integer_Array (1 .. 3);
```

(continues on next page)

(continued from previous page)

```

10  L : Natural;
11
12  Success : Boolean;
13
14  procedure Display_Add_Success (Success : Boolean) is
15  begin
16      if Success then
17          Put_Line ("Added item successfully!");
18      else
19          Put_Line ("Couldn't add item!");
20      end if;
21
22  end Display_Add_Success;
23
24  begin
25      Int_List.Init;
26
27      Int_List.Add (2, Success);
28      Display_Add_Success (Success);
29
30      Int_List.Add (5, Success);
31      Display_Add_Success (Success);
32
33      Int_List.Add (7, Success);
34      Display_Add_Success (Success);
35
36      Int_List.Add (8, Success);
37      Display_Add_Success (Success);
38
39      Int_List.Display;
40  end Test_Int;

```

Listing 2596: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Test_Int;
5
6  procedure Main is
7      type Test_Case_Index is (Int_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10     begin
11         case TC is
12             when Int_Chk =>
13                 Test_Int;
14         end case;
15     end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;

```

EXCEPTIONS

126.1 Uninitialized Value

Goal: implement an enumeration to avoid the use of uninitialized values.

Steps:

1. Implement the Options package.
 1. Declare the Option enumeration type.
 2. Declare the Uninitialized_Value exception.
 3. Implement the Image function.

Requirements:

1. Enumeration Option contains:
 1. the Uninitialized value, and
 2. the actual options:
 - Option_1,
 - Option_2,
 - Option_3.
2. Function Image returns a string for the Option type.
 1. In case the argument to Image is Uninitialized, the function must raise the Uninitialized_Value exception.

Remarks:

1. In this exercise, we employ exceptions as a mechanism to avoid the use of uninitialized values for a certain type.

Listing 2597: options.ads

```
1 package Options is
2
3   -- Declare the Option enumeration type!
4   type Option is null record;
5
6   function Image (O : Option) return String;
7
8 end Options;
```

Listing 2598: options.adb

```
1 package body Options is
2
3     function Image (O : Option) return String is
4     begin
5         return "";
6     end Image;
7
8 end Options;
```

Listing 2599: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Exceptions;   use Ada.Exceptions;
4
5 with Options;          use Options;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Options_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Check (O : Option) is
14         begin
15             Put_Line (Image (O));
16         exception
17             when E : Uninitialized_Value =>
18                 Put_Line (Exception_Message (E));
19         end Check;
20
21     begin
22         case TC is
23             when Options_Chk =>
24                 for O in Option loop
25                     Check (O);
26                 end loop;
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;
```

126.2 Numerical Exception

Goal: handle numerical exceptions in a test procedure.

Steps:

1. Add exception handling to the Check_Exception procedure.

Requirements:

1. The test procedure `Num_Exception_Test` from the `Tests` package below must be used in the implementation of `Check_Exception`.
2. The `Check_Exception` procedure must be extended to handle exceptions as follows:
 1. If the exception raised by `Num_Exception_Test` is `Constraint_Error`, the procedure must display the message "Constraint_Error detected!" to the user.
 2. Otherwise, it must display the message associated with the exception.

Remarks:

1. You can use the `Exception_Message` function to retrieve the message associated with an exception.

Listing 2600: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;
```

Listing 2601: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14        raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;
```

Listing 2602: check_exception.adb

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID) is
4 begin
5   Num_Exception_Test (ID);
6 end Check_Exception;
```

Listing 2603: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```

3  with Ada.Exceptions;    use Ada.Exceptions;
4
5  with Tests;              use Tests;
6  with Check_Exception;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16             begin
17                 Check_Exception (ID);
18             exception
19                 when Constraint_Error =>
20                     Put_Line ("Constraint_Error"
21                               & " (raised by Check_Exception) detected!");
22                 when E : others =>
23                     Put_Line (Exception_Name (E)
24                               & " (raised by Check_Exception) detected!");
25             end Check_Handle_Exception;
26
27         begin
28             case TC is
29                 when Exception_1_Chk =>
30                     Check_Handle_Exception (Test_1);
31                 when Exception_2_Chk =>
32                     Check_Handle_Exception (Test_2);
33             end case;
34         end Check;
35
36     begin
37         if Argument_Count < 1 then
38             Put_Line ("ERROR: missing arguments! Exiting...");
39             return;
40         elsif Argument_Count > 1 then
41             Put_Line ("Ignoring additional arguments...");
42         end if;
43
44         Check (Test_Case_Index'Value (Argument (1)));
45     end Main;

```

126.3 Re-raising Exceptions

Goal: make use of exception re-raising in a test procedure.

Steps:

1. Declare new exception: Another_Exception.
2. Add exception re-raise to the Check_Exception procedure.

Requirements:

1. Exception Another_Exception must be declared in the Tests package.
2. Procedure Check_Exception must be extended to *re-raise* any exception. When an exception is detected, the procedure must:
 1. display a user message (as implemented in the previous exercise), and then

2. Raise or *re-raise* exception depending on the exception that is being handled:
 1. In case of Constraint_Error exception, *re-raise* the exception.
 2. In all other cases, raise Another_Exception.

Remarks:

1. In this exercise, you should extend the implementation of the Check_Exception procedure from the previous exercise.
1. Naturally, you can use the code for the Check_Exception procedure from the previous exercise as a starting point.

Listing 2604: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;
```

Listing 2605: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14        raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;
```

Listing 2606: check_exception.ads

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID);
```

Listing 2607: check_exception.adb

```

1 procedure Check_Exception (ID : Test_ID) is
2 begin
3   Num_Exception_Test (ID);
4 end Check_Exception;
```

Listing 2608: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Exceptions;   use Ada.Exceptions;
4
5 with Tests;            use Tests;
6 with Check_Exception;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16             begin
17                 Check_Exception (ID);
18             exception
19                 when Constraint_Error =>
20                     Put_Line ("Constraint_Error"
21                               & " (raised by Check_Exception) detected!");
22                 when E : others =>
23                     Put_Line (Exception_Name (E)
24                               & " (raised by Check_Exception) detected!");
25             end Check_Handle_Exception;
26
27             begin
28                 case TC is
29                     when Exception_1_Chk =>
30                         Check_Handle_Exception (Test_1);
31                     when Exception_2_Chk =>
32                         Check_Handle_Exception (Test_2);
33                     end case;
34             end Check;
35
36 begin
37     if Argument_Count < 1 then
38         Put_Line ("ERROR: missing arguments! Exiting...");
39         return;
40     elsif Argument_Count > 1 then
41         Put_Line ("Ignoring additional arguments...");
42     end if;
43
44     Check (Test_Case_Index'Value (Argument (1)));
45 end Main;
```

TASKING

127.1 Display Service

Goal: create a simple service that displays messages to the user.

Steps:

1. Implement the `Display_Services` package.
 1. Declare the task type `Display_Service`.
 2. Implement the `Display` entry for strings.
 3. Implement the `Display` entry for integers.

Requirements:

1. Task type `Display_Service` uses the `Display` entry to display messages to the user.
2. There are two versions of the `Display` entry:
 1. One that receives messages as a string parameter.
 2. One that receives messages as an **Integer** parameter.
3. When a message is received via a `Display` entry, it must be displayed immediately to the user.

Listing 2609: `display_services.ads`

```
1 package Display_Services is
2
3 end Display_Services;
```

Listing 2610: `display_services.adb`

```
1 package body Display_Services is
2
3 end Display_Services;
```

Listing 2611: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Services; use Display_Services;
5
6 procedure Main is
7     type Test_Case_Index is (Display_Service_Chk);
8
9     procedure Check (TC : Test_Case_Index) is
10         Display : Display_Service;
```

(continues on next page)

(continued from previous page)

```

11  begin
12      case TC is
13          when Display_Service_Chk =>
14              Display.Display ("Hello");
15              delay 0.5;
16              Display.Display ("Hello again");
17              delay 0.5;
18              Display.Display (55);
19              delay 0.5;
20          end case;
21  end Check;
22
23  begin
24      if Argument_Count < 1 then
25          Put_Line ("ERROR: missing arguments! Exiting...");
26          return;
27      elsif Argument_Count > 1 then
28          Put_Line ("Ignoring additional arguments...");
29      end if;
30
31      Check (Test_Case_Index'Value (Argument (1)));
32  end Main;

```

127.2 Event Manager

Goal: implement a simple event manager.

Steps:

1. Implement the Event_Managers package.
 1. Declare the task type Event_Manager.
 2. Implement the Start entry.
 3. Implement the Event entry.

Requirements:

1. The event manager has a similar behavior as an alarm
 1. The sole purpose of this event manager is to display the event ID at the correct time.
 2. After the event ID is displayed, the task must finish.
2. The event manager (Event_Manager type) must have two entries:
 1. Start, which starts the event manager with an event ID;
 2. Event, which delays the task until a certain time and then displays the event ID as a user message.
3. The format of the user message displayed by the event manager is Event #<event_id>.
 1. You should use Natural'Image to display the ID (as indicated in the body of the Event_Managers package below).

Remarks:

1. In the Start entry, you can use the **Natural** type for the ID.
2. In the Event entry, you should use the Time type from the Ada.Real_Time package for the time parameter.

3. Note that the test application below creates an array of event managers with different delays.

Listing 2612: event_managers.ads

```

1 package Event_Managers is
2
3 end Event_Managers;
```

Listing 2613: event_managers.adb

```

1 package body Event_Managers is
2
3     -- Don't forget to display the event ID:
4     --
5     -- Put_Line ("Event #" & Natural'Image (Event_ID));
6
7 end Event_Managers;
```

Listing 2614: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Event_Managers;     use Event_Managers;
5 with Ada.Real_Time;     use Ada.Real_Time;
6
7 procedure Main is
8     type Test_Case_Index is (Event_Manager_Chk);
9
10    procedure Check (TC : Test_Case_Index) is
11        Ev_Mng : array (1 .. 5) of Event_Manager;
12    begin
13        case TC is
14            when Event_Manager_Chk =>
15                for I in Ev_Mng'Range loop
16                    Ev_Mng (I).Start (I);
17                end loop;
18                Ev_Mng (1).Event (Clock + Seconds (5));
19                Ev_Mng (2).Event (Clock + Seconds (3));
20                Ev_Mng (3).Event (Clock + Seconds (1));
21                Ev_Mng (4).Event (Clock + Seconds (2));
22                Ev_Mng (5).Event (Clock + Seconds (4));
23            end case;
24        end Check;
25
26    begin
27        if Argument_Count < 1 then
28            Put_Line ("ERROR: missing arguments! Exiting...");
29            return;
30        elsif Argument_Count > 1 then
31            Put_Line ("Ignoring additional arguments...");
32        end if;
33
34        Check (Test_Case_Index'Value (Argument (1)));
35    end Main;
```

127.3 Generic Protected Queue

Goal: create a queue container using a protected type.

Steps:

1. Implement the generic package `Gen_Queues`.
 1. Declare the protected type `Queue`.
 2. Implement the `Empty` function.
 3. Implement the `Full` function.
 4. Implement the `Push` entry.
 5. Implement the `Pop` entry.

Requirements:

1. These are the formal parameters for the generic package `Gen_Queues`:
 1. a formal modular type;
 - This modular type should be used by the `Queue` to declare an array that stores the elements of the queue.
 - The modulus of the modular type must correspond to the maximum number of elements of the queue.
 2. the data type of the elements of the queue.
 - Select a formal parameter that allows you to store elements of any data type in the queue.
2. These are the operations of the `Queue` type:
 1. Function `Empty` indicates whether the queue is empty.
 2. Function `Full` indicates whether the queue is full.
 3. Entry `Push` stores an element in the queue.
 4. Entry `Pop` removes an element from the queue and returns the element via output parameter.

Remarks:

1. In this exercise, we create a queue container by declaring and implementing a protected type (`Queue`) as part of a generic package (`Gen_Queues`).
2. As a bonus exercise, you can analyze the body of the `Queue_Tests` package and understand how the `Queue` type is used there.
 1. In particular, the procedure `Concurrent_Test` implements two tasks: `T_Producer` and `T_Consumer`. They make use of the queue concurrently.

Listing 2615: `gen_queues.ads`

```
1 package Gen_Queues is
2
3 end Gen_Queues;
```

Listing 2616: `gen_queues.adb`

```
1 package body Gen_Queues is
2
3 end Gen_Queues;
```

Listing 2617: queue_tests.ads

```

1 package Queue_Tests is
2
3     procedure Simple_Test;
4
5     procedure Concurrent_Test;
6
7 end Queue_Tests;
```

Listing 2618: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7     Max : constant := 10;
8     type Queue_Mod is mod Max;
9
10    procedure Simple_Test is
11        package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13        Q_F : Queues_Float.Queue;
14        V : Float;
15    begin
16        V := 10.0;
17        while not Q_F.Full loop
18            Q_F.Push (V);
19            V := V + 1.5;
20        end loop;
21
22        while not Q_F.Empty loop
23            Q_F.Pop (V);
24            Put_Line ("Value from queue: " & Float'Image (V));
25        end loop;
26    end Simple_Test;
27
28    procedure Concurrent_Test is
29        package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
30
31        Q_I : Queues_Integer.Queue;
32
33        task T_Producer;
34        task T_Consumer;
35
36        task body T_Producer is
37            V : Integer := 100;
38        begin
39            for I in 1 .. 2 * Max loop
40                Q_I.Push (V);
41                V := V + 1;
42            end loop;
43        end T_Producer;
44
45        task body T_Consumer is
46            V : Integer;
47        begin
48            delay 1.5;
```

(continues on next page)

(continued from previous page)

```

50     while not Q_I.Empty loop
51         Q_I.Pop (V);
52         Put_Line ("Value from queue: " & Integer'Image (V));
53         delay 0.2;
54     end loop;
55 end T_Consumer;
56 begin
57     null;
58 end Concurrent_Test;
59
60 end Queue_Tests;

```

Listing 2619: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Queue_Tests;      use Queue_Tests;
5
6  procedure Main is
7      type Test_Case_Index is (Simple_Queue_Chk,
8                               Concurrent_Queue_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11
12     begin
13         case TC is
14             when Simple_Queue_Chk =>
15                 Simple_Test;
16             when Concurrent_Queue_Chk =>
17                 Concurrent_Test;
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

DESIGN BY CONTRACTS

128.1 Price Range

Goal: use predicates to indicate the correct range of prices.

Steps:

1. Complete the Prices package.
 1. Rewrite the type declaration of Price.

Requirements:

1. Type Price must use a predicate instead of a range.

Remarks:

1. As discussed in the course, ranges are a form of contract.
 1. For example, the subtype Price below indicates that a value of this subtype must always be positive:

```
subtype Price is Amount range 0.0 .. Amount'Last;
```

2. Interestingly, you can replace ranges by predicates, which is the goal of this exercise.

Listing 2620: prices.ads

```
1 package Prices is
2
3     type Amount is delta 10.0 ** (-2) digits 12;
4
5     subtype Price is Amount range 0.0 .. Amount'Last;
6
7 end Prices;
```

Listing 2621: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Prices;             use Prices;
6
7 procedure Main is
8
9     type Test_Case_Index is
10         (Price_Range_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
```

(continues on next page)

(continued from previous page)

```

13
14     procedure Check_Range (A : Amount) is
15         P : constant Price := A;
16     begin
17         Put_Line ("Price: " & Price'Image (P));
18     end Check_Range;
19
20 begin
21     case TC is
22     when Price_Range_Chk =>
23         Check_Range (-2.0);
24     end case;
25 exception
26     when Constraint_Error =>
27         Put_Line ("Constraint_Error detected (NOT as expected).");
28     when Assert_Failure =>
29         Put_Line ("Assert_Failure detected (as expected).");
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

128.2 Pythagorean Theorem: Predicate

Goal: use the Pythagorean theorem as a predicate.

Steps:

1. Complete the Triangles package.
 1. Add a predicate to the Right_Triangle type.

Requirements:

1. The Right_Triangle type must use the Pythagorean theorem as a predicate to ensure that its components are consistent.

Remarks:

1. As you probably remember, the [Pythagoras' theorem](https://en.wikipedia.org/wiki/Pythagorean_theorem)⁵⁸⁵ states that the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides.

Listing 2622: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is record
6         H      : Length := 0;
7         -- Hypotenuse
8         C1, C2 : Length := 0;

```

(continues on next page)

⁵⁸⁵ https://en.wikipedia.org/wiki/Pythagorean_theorem

(continued from previous page)

```

9      -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2));
14
15 end Triangles;

```

Listing 2623: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 2624: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         ("      & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")");
8
9 end Triangles.IO;

```

Listing 2625: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;

```

(continues on next page)

(continued from previous page)

```

31
32   begin
33       case TC is
34           when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35           when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36           when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37           when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38           when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39           when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40       end case;
41   end Check;
42
43   begin
44       if Argument_Count < 1 then
45           Put_Line ("ERROR: missing arguments! Exiting...");
46           return;
47       elsif Argument_Count > 1 then
48           Put_Line ("Ignoring additional arguments...");
49       end if;
50
51       Check (Test_Case_Index'Value (Argument (1)));
52   end Main;

```

128.3 Pythagorean Theorem: Precondition

Goal: use the Pythagorean theorem as a precondition.

Steps:

1. Complete the Triangles package.
 1. Add a precondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a precondition to ensure that the input values are consistent.

Remarks:

1. In this exercise, you'll work again with the Right_Triangle type.
 1. This time, your job is to use a precondition instead of a predicate.
 2. The precondition is applied to the Init function, not to the Right_Triangle type.

Listing 2626: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;
9          -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2));
14
15 end Triangles;

```

Listing 2627: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 2628: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         (" " & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")");
8
9 end Triangles.IO;
```

Listing 2629: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
```

(continues on next page)

(continued from previous page)

```

40     end case;
41 end Check;
42
43 begin
44   if Argument_Count < 1 then
45     Put_Line ("ERROR: missing arguments! Exiting...");
46     return;
47   elsif Argument_Count > 1 then
48     Put_Line ("Ignoring additional arguments...");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

128.4 Pythagorean Theorem: Postcondition

Goal: use the Pythagorean theorem as a postcondition.

Steps:

1. Complete the Triangles package.
 1. Add a postcondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a postcondition to ensure that the returned object is consistent.

Remarks:

1. In this exercise, you'll work again with the Triangles package.
 1. This time, your job is to apply a postcondition instead of a precondition to the Init function.

Listing 2630: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10  end record;
11
12  function Init (H, C1, C2 : Length) return Right_Triangle is
13    ((H, C1, C2));
14
15 end Triangles;

```

Listing 2631: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 2632: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")";
8
9 end Triangles.IO;
```

Listing 2633: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43 begin
44     if Argument_Count < 1 then
45         Put_Line ("ERROR: missing arguments! Exiting...");
46         return;
47     elsif Argument_Count > 1 then
```

(continues on next page)

(continued from previous page)

```

48     Put_Line ("Ignoring additional arguments...");
49     end if;
50
51     Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

128.5 Pythagorean Theorem: Type Invariant

Goal: use the Pythagorean theorem as a type invariant.

Steps:

1. Complete the Triangles package.
 1. Add a type invariant to the Right_Triangle type.

Requirements:

1. Right_Triangle is a private type.
 1. It must use the Pythagorean theorem as a type invariant to ensure that its encapsulated components are consistent.

Remarks:

1. In this exercise, Right_Triangle is declared as a private type.
 1. In this case, we use a type invariant for Right_Triangle to check the Pythagorean theorem.
2. As a bonus, after completing the exercise, you may analyze the effect that default values have on type invariants.
 1. For example, the declaration of Right_Triangle uses zero as the default values of the three triangle lengths.
 2. If you replace those default values with Length'Last, you'll get different results.
 3. Make sure you understand why this is happening.

Listing 2634: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is private;
6
7      function Init (H, C1, C2 : Length) return Right_Triangle;
8
9  private
10
11      type Right_Triangle is record
12          H      : Length := 0;
13          -- Hypotenuse
14          C1, C2 : Length := 0;
15          -- Catheti / legs
16      end record;
17
18      function Init (H, C1, C2 : Length) return Right_Triangle is
19          ((H, C1, C2));
20
21  end Triangles;

```

Listing 2635: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 2636: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         ("(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")");
8
9 end Triangles.IO;
```

Listing 2637: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
```

(continues on next page)

(continued from previous page)

```

40     end case;
41 end Check;
42
43 begin
44   if Argument_Count < 1 then
45     Put_Line ("ERROR: missing arguments! Exiting...");
46     return;
47   elsif Argument_Count > 1 then
48     Put_Line ("Ignoring additional arguments...");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

128.6 Primary Color

Goal: extend a package for HTML colors so that it can handle primary colors.

Steps:

1. Complete the Color_Types package.
 1. Declare the HTML_RGB_Color subtype.
 2. Implement the To_Int_Color function.

Requirements:

1. The HTML_Color type is an enumeration that contains a list of HTML colors.
2. The To_RGB_Lookup_Table array implements a lookup-table to convert the colors into a hexadecimal value using RGB color components (i.e. Red, Green and Blue)
3. Function To_Int_Color extracts one of the RGB components of an HTML color and returns its hexadecimal value.
 1. The function has two parameters:
 - First parameter is the HTML color (HTML_Color type).
 - Second parameter indicates which RGB component is to be extracted from the HTML color (HTML_RGB_Color subtype).
 2. For example, if we call To_Int_Color (Salmon, Red), the function returns #FA,
 - This is the hexadecimal value of the red component of the Salmon color.
 - You can find further remarks below about this color as an example.
4. The HTML_RGB_Color subtype is limited to the primary RGB colors components (i.e. Red, Green and Blue).
 1. This subtype is used to select the RGB component in calls to To_Int_Color.
 2. You must use a predicate in the type declaration.

Remarks:

1. In this exercise, we reuse the code of the Colors: Lookup-Table exercise from the [Arrays](#) (page 2195) labs.
2. These are the hexadecimal values of the colors that we used in the original exercise:

Color	Value
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

3. You can extract the hexadecimal value of each primary color by splitting the values from the table above into three hexadecimal values with two digits each.

- For example, the hexadecimal value of Salmon is #FA8072, where:
 - the first part of this hexadecimal value (#FA) corresponds to the red component,
 - the second part (#80) corresponds to the green component, and
 - the last part (#72) corresponds to the blue component.

Listing 2638: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   subtype Int_Color is Integer range 0 .. 255;
17
18   function Image (I : Int_Color) return String;
19
20   type RGB is record
21     Red   : Int_Color;
22     Green : Int_Color;
23     Blue  : Int_Color;
24   end record;
25
26   function To_RGB (C : HTML_Color) return RGB;
27
28   function Image (C : RGB) return String;
29
30   type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32   To_RGB_Lookup_Table : constant HTML_Color_RGB_Array
33     := (Salmon      => (16#FA#, 16#80#, 16#72#),

```

(continues on next page)

(continued from previous page)

```

34      Firebrick  => (16#B2#, 16#22#, 16#22#),
35      Red        => (16#FF#, 16#00#, 16#00#),
36      Darkred    => (16#8B#, 16#00#, 16#00#),
37      Lime       => (16#00#, 16#FF#, 16#00#),
38      Forestgreen => (16#22#, 16#8B#, 16#22#),
39      Green      => (16#00#, 16#80#, 16#00#),
40      Darkgreen  => (16#00#, 16#64#, 16#00#),
41      Blue       => (16#00#, 16#00#, 16#FF#),
42      Mediumblue => (16#00#, 16#00#, 16#CD#),
43      Darkblue   => (16#00#, 16#00#, 16#8B#));
44
45      subtype HTML_RGB_Color is HTML_Color;
46
47      function To_Int_Color (C : HTML_Color;
48                             S : HTML_RGB_Color) return Int_Color;
49      -- Convert to hexadecimal value for the selected RGB component S
50
51 end Color_Types;

```

Listing 2639: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_RGB (C : HTML_Color) return RGB is
6      begin
7          return To_RGB_Lookup_Table (C);
8      end To_RGB;
9
10     function To_Int_Color (C : HTML_Color;
11                            S : HTML_RGB_Color) return Int_Color is
12     begin
13         -- Implement function!
14         return 0;
15     end To_Int_Color;
16
17     function Image (I : Int_Color) return String is
18     subtype Str_Range is Integer range 1 .. 10;
19     S : String (Str_Range);
20     begin
21         Ada.Integer_Text_IO.Put (To    => S,
22                                 Item  => I,
23                                 Base   => 16);
24         return S;
25     end Image;
26
27     function Image (C : RGB) return String is
28     begin
29         return ("(Red => " & Image (C.Red)
30                & ", Green => " & Image (C.Green)
31                & ", Blue => " & Image (C.Blue)
32                & ")");
33     end Image;
34
35 end Color_Types;

```

Listing 2640: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;

```

(continues on next page)

(continued from previous page)

```

2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Color_Types;          use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_Red_Chk,
9           HTML_Color_Green_Chk,
10          HTML_Color_Blue_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13
14          procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15              begin
16                  Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17                  for I in HTML_Color'Range loop
18                      Put_Line (HTML_Color'Image (I) & " => "
19                              & Image (To_Int_Color (I, S)) & ".");
20                  end loop;
21              end Check_HTML_Colors;
22
23          begin
24              case TC is
25                  when HTML_Color_Red_Chk =>
26                      Check_HTML_Colors (Red);
27                  when HTML_Color_Green_Chk =>
28                      Check_HTML_Colors (Green);
29                  when HTML_Color_Blue_Chk =>
30                      Check_HTML_Colors (Blue);
31              end case;
32          end Check;
33
34      begin
35          if Argument_Count < 1 then
36              Put_Line ("ERROR: missing arguments! Exiting...");
37              return;
38          elsif Argument_Count > 1 then
39              Put_Line ("Ignoring additional arguments...");
40          end if;
41
42          Check (Test_Case_Index'Value (Argument (1)));
43      end Main;

```


OBJECT-ORIENTED PROGRAMMING

129.1 Simple type extension

Goal: work with type extensions using record types containing numeric components.

Steps:

1. Implement the `Type_Extensions` package.
 1. Declare the record type `T_Float`.
 2. Declare the record type `T_Mixed`
 3. Implement the `Init` function for the `T_Float` type with a floating-point input parameter.
 4. Implement the `Init` function for the `T_Float` type with an integer input parameter.
 5. Implement the `Image` function for the `T_Float` type.
 6. Implement the `Init` function for the `T_Mixed` type with a floating-point input parameter.
 7. Implement the `Init` function for the `T_Mixed` type with an integer input parameter.
 8. Implement the `Image` function for the `T_Mixed` type.

Requirements:

1. Record type `T_Float` contains the following component:
 1. `F`, a floating-point type.
2. Record type `T_Mixed` is derived from the `T_Float` type.
 1. `T_Mixed` extends `T_Float` with the following component:
 1. `I`, an integer component.
 2. Both components must be numerically *synchronized*:
 - For example, if the floating-point component contains the value 2.0, the value of the integer component must be 2.
 - In order to simplify the implementation, you can simply use **Integer** (`F`) to convert a floating-point variable `F` to integer.
3. Function `Init` returns an object of the corresponding type (`T_Float` or `T_Mixed`).
 1. For each type, two versions of `Init` must be declared:
 1. one with a floating-point input parameter,
 2. another with an integer input parameter.
 2. The parameter to `Init` is used to initialize the record components.
4. Function `Image` returns a string for the components of the record type.

1. In case of the Image function for the T_Float type, the string must have the format "{ F => <float value> }".
 - For example, the call Image (T_Float'(Init (8.0))) should return the string "{ F => 8.00000E+00 }".
2. In case of the Image function for the T_Mixed type, the string must have the format "{ F => <float value>, I => <integer value> }".
 - For example, the call Image (T_Mixed'(Init (8.0))) should return the string "{ F => 8.00000E+00, I => 8 }".

Listing 2641: type_extensions.ads

```
1 package Type_Extensions is
2
3   -- Create declaration of T_Float type!
4   type T_Float is null record;
5
6   -- function Init ...
7
8   -- function Image ...
9
10  -- Create declaration of T_Mixed type!
11  type T_Mixed is null record;
12
13 end Type_Extensions;
```

Listing 2642: type_extensions.adb

```
1 package body Type_Extensions is
2
3 end Type_Extensions;
```

Listing 2643: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Type_Extensions; use Type_Extensions;
5
6 procedure Main is
7
8   type Test_Case_Index is
9     (Type_Extension_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    F1, F2 : T_Float;
13    M1, M2 : T_Mixed;
14  begin
15    case TC is
16    when Type_Extension_Chk =>
17      F1 := Init (2.0);
18      F2 := Init (3);
19      M1 := Init (4.0);
20      M2 := Init (5);
21
22      if M2 in T_Float'Class then
23        Put_Line ("T_Mixed is in T_Float'Class as expected");
24      end if;
25
26      Put_Line ("F1: " & Image (F1));
```

(continues on next page)

(continued from previous page)

```

27         Put_Line ("F2: " & Image (F2));
28         Put_Line ("M1: " & Image (M1));
29         Put_Line ("M2: " & Image (M2));
30     end case;
31 end Check;
32
33 begin
34     if Argument_Count < 1 then
35         Put_Line ("ERROR: missing arguments! Exiting...");
36         return;
37     elsif Argument_Count > 1 then
38         Put_Line ("Ignoring additional arguments...");
39     end if;
40
41     Check (Test_Case_Index'Value (Argument (1)));
42 end Main;

```

129.2 Online Store

Goal: create an online store for the members of an association.

Steps:

1. Implement the `Online_Store` package.
 1. Declare the `Member` type.
 2. Declare the `Full_Member` type.
 3. Implement the `Get_Status` function for the `Member` type.
 4. Implement the `Get_Price` function for the `Member` type.
 5. Implement the `Get_Status` function for the `Full_Member` type.
 6. Implement the `Get_Price` function for the `Full_Member` type.
2. Implement the `Online_Store.Tests` child package.
 1. Implement the `Simple_Test` procedure.

Requirements:

1. Package `Online_Store` implements an online store application for the members of an association.
 1. In this association, members can have one of the following status:
 - associate member, or
 - full member.
2. Function `Get_Price` returns the correct price of an item.
 1. Associate members must pay the full price when they buy items from the online store.
 2. Full members can get a discount.
 1. The discount rate can be different for each full member — depending on factors that are irrelevant for this exercise.
3. Package `Online_Store` has following types:
 1. Percentage type, which represents a percentage ranging from 0.0 to 1.0.
 2. Member type for associate members containing following components:

- Start, which indicates the starting year of the membership.
 - This information is common for both associate and full members.
 - You can use the Year_Number type from the standard Ada.Calendar package for this component.
3. Full_Member type for full members.
 1. This type must extend the Member type above.
 2. It contains the following additional component:
 - Discount, which indicates the discount rate that the full member gets in the online store.
 - This component must be of Percentage type.
 4. For the Member and Full_Member types, you must implement the following functions:
 1. Get_Status, which returns a string with the membership status.
 - The string must be "Associate Member" or "Full Member", respectively.
 2. Get_Price, which returns the *adapted price* of an item — indicating the actual due amount.
 - For example, for a full member with a 10% discount rate, the actual due amount of an item with a price of 100.00 is 90.00.
 - Associated members don't get a discount, so they always pay the full price.
 5. Procedure Simple_Test (from the Online_Store.Tests package) is used for testing.
 1. Based on a list of members that bought on the online store and the corresponding full price of the item, Simple_Test must display information about each member and the actual due amount after discounts.
 2. Information about the members must be displayed in the following format:

```
Member # <number>
Status: <status>
Since: <year>
Due Amount: <value>
-----
```

3. For this exercise, Simple_Test must use the following list:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2010	N/A	250.00
2	Full	1998	10.0 %	160.00
3	Full	1987	20.0 %	400.00
4	Associate	2013	N/A	110.00

4. In order to pass the tests, the information displayed by a call to Simple_Test must conform to the format described above.
 - You can find another example in the remarks below.

Remarks:

1. In previous labs, we could have implemented a simplified version of the system described above by simply using an enumeration type to specify the membership status. For example:

```
type Member_Status is (Associate_Member, Full_Member);
```

1. In this case, the `Get_Price` function would then evaluate the membership status and adapt the item price — assuming a fixed discount rate for all full members. This could be the corresponding function declaration:

```
type Amount is delta 10.0**(-2) digits 10;

function Get_Price (M : Member_Status;
                   P : Amount) return Amount;
```

2. In this exercise, however, we'll use type extension to represent the membership status in our application.
2. For the procedure `Simple_Test`, let's consider the following list of members as an example:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2002	N/A	100.00
2	Full	2005	10.0 %	100.00

- For this list, the test procedure displays the following information (in this exact format):

```
Member # 1
Status: Associate Member
Since: 2002
Due Amount: 100.00
-----
Member # 2
Status: Full Member
Since: 2005
Due Amount: 90.00
-----
```

- Here, although both members had the same full price (as indicated by the last column), member #2 gets a reduced due amount of 90.00 because of the full membership status.

Listing 2644: `online_store.ads`

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Online_Store is
4
5     type Amount is delta 10.0**(-2) digits 10;
6
7     subtype Percentage is Amount range 0.0 .. 1.0;
8
9     -- Create declaration of Member type!
10    --
11    -- You can use Year_Number from Ada.Calendar for the membership
12    -- starting year.
13    --
14    type Member is null record;
15
16    function Get_Status (M : Member) return String;
17
18    function Get_Price (M : Member;
19                      P : Amount) return Amount;
20
21    -- Create declaration of Full_Member type!
22    --
```

(continues on next page)

(continued from previous page)

```
23  -- Use the Percentage type for storing the membership discount.
24  --
25  type Full_Member is null record;
26
27  function Get_Status (M : Full_Member) return String;
28
29  function Get_Price (M : Full_Member;
30                    P : Amount) return Amount;
31
32  end Online_Store;
```

Listing 2645: online_store.adb

```
1  package body Online_Store is
2
3      function Get_Status (M : Member) return String is
4          ("");
5
6      function Get_Status (M : Full_Member) return String is
7          ("");
8
9      function Get_Price (M : Member;
10                       P : Amount) return Amount is (0.0);
11
12     function Get_Price (M : Full_Member;
13                       P : Amount) return Amount is
14         (0.0);
15
16  end Online_Store;
```

Listing 2646: online_store-tests.ads

```
1  package Online_Store.Tests is
2
3      procedure Simple_Test;
4
5  end Online_Store.Tests;
```

Listing 2647: online_store-tests.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Online_Store.Tests is
4
5      procedure Simple_Test is
6      begin
7          null;
8      end Simple_Test;
9
10  end Online_Store.Tests;
```

Listing 2648: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  with Online_Store; use Online_Store;
5  with Online_Store.Tests; use Online_Store.Tests;
6
7  procedure Main is
```

(continues on next page)

(continued from previous page)

```

8
9  type Test_Case_Index is
10     (Type_Chk,
11      Unit_Test_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15      function Result_Image (Result : Boolean) return String is
16          (if Result then "OK" else "not OK");
17
18  begin
19      case TC is
20      when Type_Chk =>
21          declare
22              AM : constant Member      := (Start      => 2002);
23              FM : constant Full_Member := (Start      => 1990,
24                                             Discount => 0.2);
25
26              begin
27                  Put_Line ("Testing Status of Associate Member Type => "
28                           & Result_Image (AM.Get_Status = "Associate Member"));
29                  Put_Line ("Testing Status of Full Member Type => "
30                           & Result_Image (FM.Get_Status = "Full Member"));
31                  Put_Line ("Testing Discount of Associate Member Type => "
32                           & Result_Image (AM.Get_Price (100.0) = 100.0));
33                  Put_Line ("Testing Discount of Full Member Type => "
34                           & Result_Image (FM.Get_Price (100.0) = 80.0));
35
36              end;
37              when Unit_Test_Chk =>
38                  Simple_Test;
39
40          end case;
41      end Check;
42
43  begin
44      if Argument_Count < 1 then
45          Put_Line ("ERROR: missing arguments! Exiting...");
46          return;
47      elsif Argument_Count > 1 then
48          Put_Line ("Ignoring additional arguments...");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)));
52  end Main;

```


STANDARD LIBRARY: CONTAINERS

130.1 Simple todo list

Goal: implement a simple to-do list system using vectors.

Steps:

1. Implement the `Todo_Lists` package.
 1. Declare the `Todo_Item` type.
 2. Declare the `Todo_List` type.
 3. Implement the `Add` procedure.
 4. Implement the `Display` procedure.
2. `Todo_Item` type is used to store to-do items.
 1. It should be implemented as an access type to strings.
3. `Todo_List` type is the container for all to-do items.
 1. It should be implemented as a **vector**.
4. Procedure `Add` adds items (of `Todo_Item` type) to the list (of `Todo_List` type).
 1. This requires allocating a string for the access type.
5. Procedure `Display` is used to display all to-do items.
 1. It must display one item per line.

Remarks:

1. This exercise is based on the *Simple todo list* exercise from the [More About Types](#) (page 2213).
 1. Your goal is to rewrite that exercise using vectors instead of arrays.
 2. You may reuse the code you've already implemented as a starting point.

Listing 2649: `todo_lists.ads`

```
1 package Todo_Lists is
2
3   type Todo_Item is access String;
4
5   type Todo_List is null record;
6
7   procedure Add (Todos : in out Todo_List;
8                 Item   : String);
9
10  procedure Display (Todos : Todo_List);
11
12 end Todo_Lists;
```


Listing 2650: todo_lists.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5     procedure Add (Todos : in out Todo_List;
6                   Item  : String) is
7     begin
8         null;
9     end Add;
10
11    procedure Display (Todos : Todo_List) is
12    begin
13        Put_Line ("TO-DO LIST");
14    end Display;
15
16 end Todo_Lists;
```

Listing 2651: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Todo_Lists;        use Todo_Lists;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Todo_List_Chk);
9
10    procedure Check (TC : Test_Case_Index) is
11    T : Todo_List;
12    begin
13        case TC is
14        when Todo_List_Chk =>
15            Add (T, "Buy milk");
16            Add (T, "Buy tea");
17            Add (T, "Buy present");
18            Add (T, "Buy tickets");
19            Add (T, "Pay electricity bill");
20            Add (T, "Schedule dentist appointment");
21            Add (T, "Call sister");
22            Add (T, "Revise spreadsheet");
23            Add (T, "Edit entry page");
24            Add (T, "Select new design");
25            Add (T, "Create upgrade plan");
26            Display (T);
27        end case;
28    end Check;
29
30    begin
31        if Argument_Count < 1 then
32            Put_Line ("ERROR: missing arguments! Exiting...");
33            return;
34        elsif Argument_Count > 1 then
35            Put_Line ("Ignoring additional arguments...");
36        end if;
37
38        Check (Test_Case_Index'Value (Argument (1)));
39    end Main;
```

130.2 List of unique integers

Goal: create function that removes duplicates from and orders a collection of elements.

Steps:

1. Implement package `Ops`.
 1. Declare the `Int_Array` type.
 2. Declare the `Integer_Sets` type.
 3. Implement the `Get_Unique` function that returns a set.
 4. Implement the `Get_Unique` function that returns an array of integer values.

Requirements:

1. The `Int_Array` type is an unconstrained array of positive range.
2. The `Integer_Sets` package is an instantiation of the `Ordered_Sets` package for the **Integer** type.
3. The `Get_Unique` function must remove duplicates from an input array of integer values and order the elements.
 1. For example:
 - if the input array contains `(7, 7, 1)`
 - the function must return `(1, 7)`.
 2. You must implement this function by using sets from the `Ordered_Sets` package.
 3. `Get_Unique` must be implemented in two versions:
 - one version that returns a set — `Set` type from the `Ordered_Sets` package.
 - one version that returns an array of integer values — `Int_Array` type.

Remarks:

1. Sets — as the one found in the generic `Ordered_Sets` package — are useful for quickly and easily creating an algorithm that removes duplicates from a list of elements.

Listing 2652: ops.ads

```

1 with Ada.Containers.Ordered_Sets;
2
3 package Ops is
4
5     -- type Int_Array is ...
6
7     -- package Integer_Sets is ...
8
9     subtype Int_Set is Integer_Sets.Set;
10
11     function Get_Unique (A : Int_Array) return Int_Set;
12
13     function Get_Unique (A : Int_Array) return Int_Array;
14
15 end Ops;
```

Listing 2653: ops.adb

```

1 package body Ops is
2
3     function Get_Unique (A : Int_Array) return Int_Set is
```

(continues on next page)

(continued from previous page)

```

4   begin
5       null;
6   end Get_Unique;
7
8   function Get_Unique (A : Int_Array) return Int_Array is
9   begin
10      null;
11  end Get_Unique;
12
13 end Ops;

```

Listing 2654: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Ops;                   use Ops;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Get_Unique_Set_Chk,
9           Get_Unique_Array_Chk);
10
11     procedure Check (TC : Test_Case_Index;
12                     A : Int_Array) is
13
14         procedure Display_Unique_Set (A : Int_Array) is
15             S : constant Int_Set := Get_Unique (A);
16         begin
17             for E of S loop
18                 Put_Line (Integer'Image (E));
19             end loop;
20         end Display_Unique_Set;
21
22         procedure Display_Unique_Array (A : Int_Array) is
23             AU : constant Int_Array := Get_Unique (A);
24         begin
25             for E of AU loop
26                 Put_Line (Integer'Image (E));
27             end loop;
28         end Display_Unique_Array;
29
30     begin
31         case TC is
32             when Get_Unique_Set_Chk => Display_Unique_Set (A);
33             when Get_Unique_Array_Chk => Display_Unique_Array (A);
34         end case;
35     end Check;
36
37 begin
38     if Argument_Count < 3 then
39         Put_Line ("ERROR: missing arguments! Exiting...");
40         return;
41     else
42         declare
43             A : Int_Array (1 .. Argument_Count - 1);
44         begin
45             for I in A'Range loop
46                 A (I) := Integer'Value (Argument (1 + I));
47             end loop;
48             Check (Test_Case_Index'Value (Argument (1)), A);

```

(continues on next page)

(continued from previous page)

```
49     end;  
50   end if;  
51 end Main;
```


STANDARD LIBRARY: DATES & TIMES

131.1 Holocene calendar

Goal: create a function that returns the year in the Holocene calendar.

Steps:

1. Implement the `To_Holocene_Year` function.

Requirements:

1. The `To_Holocene_Year` extracts the year from a time object (Time type) and returns the corresponding year for the [Holocene calendar](https://en.wikipedia.org/wiki/Holocene_calendar)⁵⁸⁶.
 1. For positive (AD) years, the Holocene year is calculated by adding 10,000 to the year number.

Remarks:

1. In this exercise, we don't deal with BC years.
2. Note that the year component of the Time type from the `Ada.Calendar` package is limited to years starting with 1901.

Listing 2655: `to_holocene_year.adb`

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 function To_Holocene_Year (T : Time) return Integer is
4 begin
5     return 0;
6 end To_Holocene_Year;
```

Listing 2656: `main.adb`

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4
5 with To_Holocene_Year;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Holocene_Chk);
10
11     procedure Display_Holocene_Year (Y : Year_Number) is
12         HY : Integer;
13     begin
14         HY := To_Holocene_Year (Time_Of (Y, 1, 1));
```

(continues on next page)

⁵⁸⁶ https://en.wikipedia.org/wiki/Holocene_calendar

(continued from previous page)

```

15     Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
16     Put_Line ("Year (Holocene): " & Integer'Image (HY));
17 end Display_Holocene_Year;
18
19 procedure Check (TC : Test_Case_Index) is
20 begin
21     case TC is
22     when Holocene_Chk =>
23         Display_Holocene_Year (2012);
24         Display_Holocene_Year (2020);
25     end case;
26 end Check;
27
28 begin
29     if Argument_Count < 1 then
30         Put_Line ("ERROR: missing arguments! Exiting...");
31         return;
32     elsif Argument_Count > 1 then
33         Put_Line ("Ignoring additional arguments...");
34     end if;
35
36     Check (Test_Case_Index'Value (Argument (1)));
37 end Main;

```

131.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item type.
 2. Declare the Event_Items type.
2. Implement the Events.Lists package.
 1. Declare the Event_List type.
 2. Implement the Add procedure.
 3. Implement the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description shall be stored in an access-to-string type.
2. The Event_Items type stores a list of events.
 1. This will be used later to represent multiple events for a specific date.
 2. You shall use a vector for this type.
3. The Events.Lists package contains the subprograms that are used in the test application.
4. The Event_List type (from the Events.Lists package) maps a list of events to a specific date.
 1. You must use the Event_Items type for the list of events.
 2. You shall use the Time type from the Ada.Calendar package for the dates.

3. Since we expect the events to be ordered by the date, you shall use ordered maps for the Event_List type.
5. Procedure Add adds an event into the list of events for a specific date.
6. Procedure Display must display all events for each date (ordered by date) using the following format:

```
<event_date #1>
  <description of item #1a>
  <description of item #1b>
<event_date #2>
  <description of item #2a>
  <description of item #2b>
```

1. You should use the auxiliary Date_Image function — available in the body of the Events.Lists package — to display the date in the YYYY-MM-DD format.

Remarks:

1. Let's briefly illustrate the expected output of this system.

1. Consider the following example:

```
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

with Events.Lists;           use Events.Lists;

procedure Test is
  EL : Event_List;
begin
  EL.Add (Time_Of (2019, 4, 16),
    "Item #2");
  EL.Add (Time_Of (2019, 4, 15),
    "Item #1");
  EL.Add (Time_Of (2019, 4, 16),
    "Item #3");
  EL.Display;
end Test;
```

2. The expected output of the Test procedure must be:

```
EVENTS LIST
- 2019-04-15
  - Item #1
- 2019-04-16
  - Item #2
  - Item #3
```

Listing 2657: events.ads

```
1 package Events is
2
3   type Event_Item is null record;
4
5   type Event_Items is null record;
6
7 end Events;
```


Listing 2658: events-lists.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Events.Lists is
4
5     type Event_List is tagged private;
6
7     procedure Add (Events      : in out Event_List;
8                   Event_Time  : Time;
9                   Event       : String);
10
11     procedure Display (Events : Event_List);
12
13 private
14
15     type Event_List is tagged null record;
16
17 end Events.Lists;
```

Listing 2659: events-lists.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events      : in out Event_List;
7                   Event_Time  : Time;
8                   Event       : String) is
9
10     begin
11         null;
12     end Add;
13
14     function Date_Image (T : Time) return String is
15         Date_Img : constant String := Image (T);
16     begin
17         return Date_Img (1 .. 10);
18     end;
19
20     procedure Display (Events : Event_List) is
21         T : Time;
22     begin
23         Put_Line ("EVENTS LIST");
24         -- You should use Date_Image (T) here!
25     end Display;
26
27 end Events.Lists;
```

Listing 2660: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6 with Events.Lists;           use Events.Lists;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Event_List_Chk);
```

(continues on next page)

(continued from previous page)

```
11
12 procedure Check (TC : Test_Case_Index) is
13   EL : Event_List;
14 begin
15   case TC is
16     when Event_List_Chk =>
17       EL.Add (Time_Of (2018, 2, 16),
18         "Final check");
19       EL.Add (Time_Of (2018, 2, 16),
20         "Release");
21       EL.Add (Time_Of (2018, 12, 3),
22         "Brother's birthday");
23       EL.Add (Time_Of (2018, 1, 1),
24         "New Year's Day");
25       EL.Display;
26   end case;
27 end Check;
28
29 begin
30   if Argument_Count < 1 then
31     Put_Line ("ERROR: missing arguments! Exiting...");
32     return;
33   elsif Argument_Count > 1 then
34     Put_Line ("Ignoring additional arguments...");
35   end if;
36
37   Check (Test_Case_Index'Value (Argument (1)));
38 end Main;
```


STANDARD LIBRARY: STRINGS

132.1 Concatenation

Goal: implement functions to concatenate an array of unbounded strings.

Steps:

1. Implement the `Str_Concat` package.
 1. Implement the `Concat` function for `Unbounded_String`.
 2. Implement the `Concat` function for **String**.

Requirements:

1. The first `Concat` function receives an unconstrained array of unbounded strings and returns the concatenation of those strings as an unbounded string.
 1. The second `Concat` function has the same parameters, but returns a standard string (**String** type).
2. Both `Concat` functions have the following parameters:
 1. An unconstrained array of `Unbounded_String` strings (`Unbounded_Strings` type).
 2. `Trim_Str`, a Boolean parameter indicating whether each unbounded string must be trimmed.
 3. `Add_Whitespace`, a Boolean parameter indicating whether a whitespace shall be added between each unbounded string and the next one.
 1. No whitespace shall be added after the last string of the array.

Remarks:

1. You can use the `Trim` function from the `Ada.Strings.Unbounded` package.

Listing 2661: `str_concat.ads`

```
1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3 package Str_Concat is
4
5     type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7     function Concat (USA           : Unbounded_Strings;
8                     Trim_Str      : Boolean;
9                     Add_Whitespace : Boolean) return Unbounded_String;
10
11     function Concat (USA           : Unbounded_Strings;
12                     Trim_Str      : Boolean;
13                     Add_Whitespace : Boolean) return String;
14
15 end Str_Concat;
```

Listing 2662: str_concat.adb

```

1  with Ada.Strings; use Ada.Strings;
2
3  package body Str_Concat is
4
5      function Concat (USA           : Unbounded_Strings;
6                      Trim_Str      : Boolean;
7                      Add_Whitespace : Boolean) return Unbounded_String is
8
9          begin
10             return "";
11         end Concat;
12
13         function Concat (USA           : Unbounded_Strings;
14                         Trim_Str      : Boolean;
15                         Add_Whitespace : Boolean) return String is
16
17             begin
18                 return "";
19             end Concat;
20
21     end Str_Concat;

```

Listing 2663: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
4
5  with Str_Concat; use Str_Concat;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Unbounded_Concat_No_Trim_No_WS_Chk,
10           Unbounded_Concat_Trim_No_WS_Chk,
11           String_Concat_Trim_WS_Chk,
12           Concat_Single_Element);
13
14      procedure Check (TC : Test_Case_Index) is
15      begin
16          case TC is
17              when Unbounded_Concat_No_Trim_No_WS_Chk =>
18                  declare
19                      S : constant Unbounded_Strings := (
20                          To_Unbounded_String ("Hello"),
21                          To_Unbounded_String (" World"),
22                          To_Unbounded_String ("!"));
23                  begin
24                      Put_Line (To_String (Concat (S, False, False)));
25                  end;
26              when Unbounded_Concat_Trim_No_WS_Chk =>
27                  declare
28                      S : constant Unbounded_Strings := (
29                          To_Unbounded_String (" This "),
30                          To_Unbounded_String (" _is_ "),
31                          To_Unbounded_String (" a "),
32                          To_Unbounded_String (" _check "));
33                  begin
34                      Put_Line (To_String (Concat (S, True, False)));
35                  end;
36              when String_Concat_Trim_WS_Chk =>
37                  declare

```

(continues on next page)

(continued from previous page)

```

38         S : constant Unbounded_Strings := (
39             To_Unbounded_String (" This "),
40             To_Unbounded_String (" is a "),
41             To_Unbounded_String (" test. "));
42     begin
43         Put_Line (Concat (S, True, True));
44     end;
45     when Concat_Single_Element =>
46     declare
47         S : constant Unbounded_Strings := (
48             1 => To_Unbounded_String (" Hi "));
49     begin
50         Put_Line (Concat (S, True, True));
51     end;
52     end case;
53 end Check;
54
55 begin
56     if Argument_Count < 1 then
57         Put_Line ("ERROR: missing arguments! Exiting...");
58         return;
59     elsif Argument_Count > 1 then
60         Put_Line ("Ignoring additional arguments...");
61     end if;
62
63     Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

132.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item subtype.
2. Implement the Events.Lists package.
 1. Adapt the Add procedure.
 2. Adapt the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description is declared as a subtype of unbounded string.
2. Procedure Add adds an event into the list of events for a specific date.
 1. The declaration of E needs to be adapted to use unbounded strings.
3. Procedure Display must display all events for each date (ordered by date) using the following format:
 1. The arguments to Put_Line need to be adapted to use unbounded strings.

Remarks:

1. We use the lab on the list of events from the previous chapter (*Standard library: Dates & Times* (page 2283)) as a starting point.

Listing 2664: events.ads

```
1 with Ada.Containers.Vectors;
2
3 package Events is
4
5     -- subtype Event_Item is
6
7     package Event_Item_Containers is new
8         Ada.Containers.Vectors
9         (Index_Type => Positive,
10          Element_Type => Event_Item);
11
12     subtype Event_Items is Event_Item_Containers.Vector;
13
14 end Events;
```

Listing 2665: events-lists.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2 with Ada.Containers.Ordered_Maps;
3
4 package Events.Lists is
5
6     type Event_List is tagged private;
7
8     procedure Add (Events : in out Event_List;
9                  Event_Time : Time;
10                 Event : String);
11
12     procedure Display (Events : Event_List);
13
14 private
15
16     package Event_Time_Item_Containers is new
17         Ada.Containers.Ordered_Maps
18         (Key_Type => Time,
19          Element_Type => Event_Items,
20          "=" => Event_Item_Containers "=");
21
22     type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;
```

Listing 2666: events-lists.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events : in out Event_List;
7                  Event_Time : Time;
8                  Event : String) is
9         use Event_Item_Containers;
10        E : constant Event_Item := new String'(Event);
11    begin
12        if not Events.Contains (Event_Time) then
13            Events.Include (Event_Time, Empty_Vector);
14        end if;
15        Events (Event_Time).Append (E);
```

(continues on next page)

(continued from previous page)

```

16  end Add;
17
18  function Date_Image (T : Time) return String is
19      Date_Img : constant String := Image (T);
20  begin
21      return Date_Img (1 .. 10);
22  end;
23
24  procedure Display (Events : Event_List) is
25      use Event_Time_Item_Containers;
26      T : Time;
27  begin
28      Put_Line ("EVENTS LIST");
29      for C in Events.Iterate loop
30          T := Key (C);
31          Put_Line ("- " & Date_Image (T));
32          for I of Events (C) loop
33              Put_Line ("    - " & I.all);
34          end loop;
35      end loop;
36  end Display;
37
38  end Events.Lists;

```

Listing 2667: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;          use Events.Lists;
9
10 procedure Main is
11     type Test_Case_Index is
12         (Unbounded_String_Chk,
13          Event_List_Chk);
14
15     procedure Check (TC : Test_Case_Index) is
16         EL : Event_List;
17     begin
18         case TC is
19             when Unbounded_String_Chk =>
20                 declare
21                     S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22                 begin
23                     Put_Line (To_String (S));
24                 end;
25             when Event_List_Chk =>
26                 EL.Add (Time_Of (2018, 2, 16),
27                     "Final check");
28                 EL.Add (Time_Of (2018, 2, 16),
29                     "Release");
30                 EL.Add (Time_Of (2018, 12, 3),
31                     "Brother's birthday");
32                 EL.Add (Time_Of (2018, 1, 1),
33                     "New Year's Day");
34                 EL.Display;
35             end case;

```

(continues on next page)

(continued from previous page)

```
36     end Check;
37
38 begin
39     if Argument_Count < 1 then
40         Put_Line ("ERROR: missing arguments! Exiting...");
41         return;
42     elsif Argument_Count > 1 then
43         Put_Line ("Ignoring additional arguments...");
44     end if;
45
46     Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

STANDARD LIBRARY: NUMERICS

133.1 Decibel Factor

Goal: implement functions to convert from Decibel values to factors and vice-versa.

Steps:

1. Implement the Decibels package.
 1. Implement the To_Decibel function.
 2. Implement the To_Factor function.

Requirements:

1. The subtypes Decibel and Factor are based on a floating-point type.
2. Function To_Decibel converts a multiplication factor (or ratio) to decibels.
 - For the implementation, use $20 * \log_{10}(F)$, where F is the factor/ratio.
3. Function To_Factor converts a value in decibels to a multiplication factor (or ratio).
 - For the implementation, use $10^{D/20}$, where D is the value in Decibel.

Remarks:

1. The [Decibel](https://en.wikipedia.org/wiki/Decibel)⁵⁸⁷ is used to express the ratio of two values on a logarithmic scale.
 1. For example, an increase of 6 dB corresponds roughly to a multiplication by two (or an increase by 100 % of the original value).
2. You can find the functions that you'll need for the calculation in the Ada.Numerics.Elementary_Functions package.

Listing 2668: decibels.ads

```
1 package Decibels is
2
3     subtype Decibel is Float;
4     subtype Factor is Float;
5
6     function To_Decibel (F : Factor) return Decibel;
7
8     function To_Factor (D : Decibel) return Factor;
9
10 end Decibels;
```

⁵⁸⁷ <https://en.wikipedia.org/wiki/Decibel>

Listing 2669: decibels.adb

```

1 package body Decibels is
2
3     function To_Decibel (F : Factor) return Decibel is
4     begin
5         return 0.0;
6     end To_Decibel;
7
8     function To_Factor (D : Decibel) return Factor is
9     begin
10        return 0.0;
11    end To_Factor;
12
13 end Decibels;

```

Listing 2670: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Decibels;          use Decibels;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Db_Chk,
9          Factor_Chk);
10
11     procedure Check (TC : Test_Case_Index; V : Float) is
12
13         package F_IO is new Ada.Text_IO.Float_IO (Factor);
14         package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16         procedure Put_Decibel_Cnvt (D : Decibel) is
17             F : constant Factor := To_Factor (D);
18         begin
19             D_IO.Put (D, 0, 2, 0);
20             Put (" dB => Factor of ");
21             F_IO.Put (F, 0, 2, 0);
22             New_Line;
23         end;
24
25         procedure Put_Factor_Cnvt (F : Factor) is
26             D : constant Decibel := To_Decibel (F);
27         begin
28             Put ("Factor of ");
29             F_IO.Put (F, 0, 2, 0);
30             Put (" => ");
31             D_IO.Put (D, 0, 2, 0);
32             Put_Line (" dB");
33         end;
34     begin
35         case TC is
36             when Db_Chk =>
37                 Put_Decibel_Cnvt (Decibel (V));
38             when Factor_Chk =>
39                 Put_Factor_Cnvt (Factor (V));
40         end case;
41     end Check;
42
43 begin
44     if Argument_Count < 2 then

```

(continues on next page)

(continued from previous page)

```

45     Put_Line ("ERROR: missing arguments! Exiting...");
46     return;
47   elsif Argument_Count > 2 then
48     Put_Line ("Ignoring additional arguments...");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52 end Main;

```

133.2 Root-Mean-Square

Goal: implement a function to calculate the root-mean-square of a sequence of values.

Steps:

1. Implement the Signals package.
 1. Implement the Rms function.

Requirements:

1. Subtype Sig_Value is based on a floating-point type.
2. Type Signal is an unconstrained array of Sig_Value elements.
3. Function Rms calculates the RMS of a sequence of values stored in an array of type Signal.
 1. See the remarks below for a description of the RMS calculation.

Remarks:

1. The [root-mean-square](https://en.wikipedia.org/wiki/Root_mean_square)⁵⁸⁸ (RMS) value is an important information associated with sequences of values.
 1. It's used, for example, as a measurement for signal processing.
 2. It is calculated by:
 1. Creating a sequence S with the square of each value of an input sequence S_{in} .
 2. Calculating the mean value M of the sequence S .
 3. Calculating the square-root R of M .
 3. You can optimize the algorithm above by combining steps #1 and #2 into a single step.

Listing 2671: signals.ads

```

1 package Signals is
2
3   subtype Sig_Value is Float;
4
5   type Signal is array (Natural range <>) of Sig_Value;
6
7   function Rms (S : Signal) return Sig_Value;
8
9 end Signals;

```

⁵⁸⁸ https://en.wikipedia.org/wiki/Root_mean_square

Listing 2672: signals.adb

```

1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Signals is
4
5     function Rms (S : Signal) return Sig_Value is
6     begin
7         return 0.0;
8     end;
9
10 end Signals;
```

Listing 2673: signals-std.ads

```

1 package Signals.Std is
2
3     Sample_Rate : Float := 8000.0;
4
5     function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7     function Generate_Square (N : Positive) return Signal;
8
9     function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;
```

Listing 2674: signals-std.adb

```

1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4 package body Signals.Std is
5
6     function Generate_Sine (N : Positive; Freq : Float) return Signal is
7     S : Signal (0 .. N - 1);
8     begin
9         for I in S'First .. S'Last loop
10             S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11         end loop;
12
13         return S;
14     end;
15
16     function Generate_Square (N : Positive) return Signal is
17     S : constant Signal (0 .. N - 1) := (others => 1.0);
18     begin
19         return S;
20     end;
21
22     function Generate_Triangular (N : Positive) return Signal is
23     S : Signal (0 .. N - 1);
24     S_Half : constant Natural := S'Last / 2;
25     begin
26         for I in S'First .. S_Half loop
27             S (I) := 1.0 * (Float (I) / Float (S_Half));
28         end loop;
29         for I in S_Half .. S'Last loop
30             S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31         end loop;
32
33     end;
```

(continues on next page)

(continued from previous page)

```

33     return S;
34 end;
35
36 end Signals.Std;

```

Listing 2675: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Signals;               use Signals;
5  with Signals.Std;           use Signals.Std;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Sine_Signal_Chk,
10           Square_Signal_Chk,
11           Triangular_Signal_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
15
16          N : constant Positive := 1024;
17          S_Si : constant Signal := Generate_Sine (N, 440.0);
18          S_Sq : constant Signal := Generate_Square (N);
19          S_Tr : constant Signal := Generate_Triangular (N + 1);
20      begin
21          case TC is
22              when Sine_Signal_Chk =>
23                  Put ("RMS of Sine Signal: ");
24                  Sig_IO.Put (Rms (S_Si), 0, 2, 0);
25                  New_Line;
26              when Square_Signal_Chk =>
27                  Put ("RMS of Square Signal: ");
28                  Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
29                  New_Line;
30              when Triangular_Signal_Chk =>
31                  Put ("RMS of Triangular Signal: ");
32                  Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
33                  New_Line;
34          end case;
35      end Check;
36
37      begin
38          if Argument_Count < 1 then
39              Put_Line ("ERROR: missing arguments! Exiting...");
40              return;
41          elsif Argument_Count > 1 then
42              Put_Line ("Ignoring additional arguments...");
43          end if;
44
45          Check (Test_Case_Index'Value (Argument (1)));
46      end Main;

```

133.3 Rotation

Goal: use complex numbers to calculate the positions of an object in a circle after rotation.

Steps:

1. Implement the `Rotation` package.
 1. Implement the `Rotation` function.

Requirements:

1. Type `Complex_Points` is an unconstrained array of complex values.
2. Function `Rotation` returns a list of positions (represented by the `Complex_Points` type) when dividing a circle in N equal slices.
 1. See the remarks below for a more detailed explanation.
 2. You must use functions from `Ada.Numerics.Complex_Types` to implement `Rotation`.
3. Subtype `Angle` is based on a floating-point type.
4. Type `Angles` is an unconstrained array of angles.
5. Function `To_Angles` returns a list of angles based on an input list of positions.

Remarks:

1. Complex numbers are particularly useful in computer graphics to simplify the calculation of rotations.
 1. For example, let's assume you've drawn an object on your screen on position (1.0, 0.0).
 2. Now, you want to move this object in a circular path — i.e. make it rotate around position (0.0, 0.0) on your screen.
 - You could use *sine* and *cosine* functions to calculate each position of the path.
 - However, you could also calculate the positions using complex numbers.
2. In this exercise, you'll use complex numbers to calculate the positions of an object that starts on zero degrees — on position (1.0, 0.0) — and rotates around (0.0, 0.0) for N slices of a circle.
 1. For example, if we divide the circle in four slices, the object's path will consist of following points / positions:

```
Point #1: ( 1.0,  0.0)
Point #2: ( 0.0,  1.0)
Point #3: (-1.0,  0.0)
Point #4: ( 0.0, -1.0)
Point #5: ( 1.0,  0.0)
```

1. As expected, point #5 is equal to the starting point (point #1), since the object rotates around (0.0, 0.0) and returns to the starting point.
2. We can also describe this path in terms of angles. The following list presents the angles for the path on a four-sliced circle:

```
Point #1:    0.00 degrees
Point #2:   90.00 degrees
Point #3:  180.00 degrees
Point #4: -90.00 degrees (= 270 degrees)
Point #5:    0.00 degrees
```

1. To rotate a complex number simply multiply it by a unit vector whose arg is the radian angle to be rotated: $Z = e^{\frac{2\pi}{N}}$

Listing 2676: rotation.ads

```

1 with Ada.Numerics.Complex_Types;
2 use Ada.Numerics.Complex_Types;
3
4 package Rotation is
5
6     type Complex_Points is array (Positive range <>) of Complex;
7
8     function Rotation (N : Positive) return Complex_Points;
9
10 end Rotation;

```

Listing 2677: rotation.adb

```

1 with Ada.Numerics; use Ada.Numerics;
2
3 package body Rotation is
4
5     function Rotation (N : Positive) return Complex_Points is
6         C : Complex_Points (1 .. 1) := (others => (0.0, 0.0));
7     begin
8         return C;
9     end;
10
11 end Rotation;

```

Listing 2678: angles.ads

```

1 with Rotation; use Rotation;
2
3 package Angles is
4
5     subtype Angle is Float;
6
7     type Angles is array (Positive range <>) of Angle;
8
9     function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;

```

Listing 2679: angles.adb

```

1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4 package body Angles is
5
6     function To_Angles (C : Complex_Points) return Angles is
7     begin
8         return A : Angles (C'Range) do
9             for I in A'Range loop
10                 A (I) := Argument (C (I)) / Pi * 180.0;
11             end loop;
12         end return;
13     end To_Angles;
14
15 end Angles;

```


Listing 2680: rotation-tests.ads

```

1 package Rotation.Tests is
2
3     procedure Test_Rotation (N : Positive);
4
5     procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;
```

Listing 2681: rotation-tests.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3 with Ada.Numerics;         use Ada.Numerics;
4
5 with Angles;               use Angles;
6
7 package body Rotation.Tests is
8
9     package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10    package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12    --
13    -- Adapt value due to floating-point inaccuracies
14    --
15
16    function Adapt (C : Complex) return Complex is
17        function Check_Zero (F : Float) return Float is
18            (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19        begin
20            return C_Out : Complex := C do
21                C_Out.Re := Check_Zero (C_Out.Re);
22                C_Out.Im := Check_Zero (C_Out.Im);
23            end return;
24    end Adapt;
25
26    function Adapt (A : Angle) return Angle is
27        (if A <= -179.99 and A >= -180.01 then 180.0 else A);
28
29    procedure Test_Rotation (N : Positive) is
30        C : constant Complex_Points := Rotation (N);
31    begin
32        Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
33        for V of C loop
34            Put ("Point: ");
35            C_IO.Put (Adapt (V), 0, 1, 0);
36            New_Line;
37        end loop;
38    end Test_Rotation;
39
40    procedure Test_Angles (N : Positive) is
41        C : constant Complex_Points := Rotation (N);
42        A : constant Angles.Angles := To_Angles (C);
43    begin
44        Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
45        for V of A loop
46            Put ("Angle: ");
47            F_IO.Put (Adapt (V), 0, 2, 0);
48            Put_Line (" degrees");
49        end loop;
```

(continues on next page)

(continued from previous page)

```

50     end Test_Angles;
51
52 end Rotation.Tests;

```

Listing 2682: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Rotation.Tests;        use Rotation.Tests;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Rotation_Chk,
9           Angles_Chk);
10
11     procedure Check (TC : Test_Case_Index; N : Positive) is
12     begin
13         case TC is
14             when Rotation_Chk =>
15                 Test_Rotation (N);
16             when Angles_Chk =>
17                 Test_Angles (N);
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 2 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 2 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30 end Main;

```


SOLUTIONS

134.1 Imperative Language

134.1.1 Hello World

Listing 2683: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     Put_Line ("Hello World!");
6 end Main;
```

134.1.2 Greetings

Listing 2684: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Greet (Name : String) is
7     begin
8         Put_Line ("Hello " & Name & "!");
9     end Greet;
10
11 begin
12     if Argument_Count < 1 then
13         Put_Line ("ERROR: missing arguments! Exiting...");
14         return;
15     elsif Argument_Count > 1 then
16         Put_Line ("Ignoring additional arguments...");
17     end if;
18
19     Greet (Argument (1));
20 end Main;
```

134.1.3 Positive Or Negative

Listing 2685: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 2686: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     if X > 0 then
6         Put_Line ("Positive");
7     elsif X < 0 then
8         Put_Line ("Negative");
9     else
10        Put_Line ("Zero");
11    end if;
12 end Classify_Number;
```

Listing 2687: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Classify_Number;
5
6 procedure Main is
7     A : Integer;
8 begin
9     if Argument_Count < 1 then
10        Put_Line ("ERROR: missing arguments! Exiting...");
11        return;
12    elsif Argument_Count > 1 then
13        Put_Line ("Ignoring additional arguments...");
14    end if;
15
16    A := Integer'Value (Argument (1));
17
18    Classify_Number (A);
19 end Main;
```

134.1.4 Numbers

Listing 2688: display_numbers.ads

```
1 procedure Display_Numbers (A, B : Integer);
```

Listing 2689: display_numbers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Numbers (A, B : Integer) is
4     X, Y : Integer;
5 begin
6     if A <= B then
7         X := A;
8         Y := B;
9     else
10        X := B;
11        Y := A;
12    end if;
13
14    for I in X .. Y loop
```

(continues on next page)

(continued from previous page)

```

15     Put_Line (Integer'Image (I));
16 end loop;
17 end Display_Numbers;

```

Listing 2690: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Display_Numbers;
5
6  procedure Main is
7      A, B : Integer;
8  begin
9      if Argument_Count < 2 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 2 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     A := Integer'Value (Argument (1));
17     B := Integer'Value (Argument (2));
18
19     Display_Numbers (A, B);
20 end Main;

```

134.2 Subprograms

134.2.1 Subtract Procedure

Listing 2691: subtract.ads

```

1  procedure Subtract (A, B : Integer;
2                      Result : out Integer);

```

Listing 2692: subtract.adb

```

1  procedure Subtract (A, B : Integer;
2                      Result : out Integer) is
3  begin
4      Result := A - B;
5  end Subtract;

```

Listing 2693: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Subtract;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Sub_10_1_Chk,
9           Sub_10_100_Chk,
10          Sub_0_5_Chk,
11          Sub_0_Minus_5_Chk);

```

(continues on next page)

(continued from previous page)

```

12
13 procedure Check (TC : Test_Case_Index) is
14     Result : Integer;
15 begin
16     case TC is
17     when Sub_10_1_Chk =>
18         Subtract (10, 1, Result);
19         Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21         Subtract (10, 100, Result);
22         Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Subtract (0, 5, Result);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Subtract (0, -5, Result);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

134.2.2 Subtract Function

Listing 2694: subtract.ads

```

1 function Subtract (A, B : Integer) return Integer;

```

Listing 2695: subtract.adb

```

1 function Subtract (A, B : Integer) return Integer is
2 begin
3     return A - B;
4 end Subtract;

```

Listing 2696: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Subtract;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Sub_10_1_Chk,
9          Sub_10_100_Chk,
10         Sub_0_5_Chk,
11         Sub_0_Minus_5_Chk);
12
13     procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

14     Result : Integer;
15 begin
16     case TC is
17     when Sub_10_1_Chk =>
18         Result := Subtract (10, 1);
19         Put_Line ("Result: " & Integer'Image (Result));
20     when Sub_10_100_Chk =>
21         Result := Subtract (10, 100);
22         Put_Line ("Result: " & Integer'Image (Result));
23     when Sub_0_5_Chk =>
24         Result := Subtract (0, 5);
25         Put_Line ("Result: " & Integer'Image (Result));
26     when Sub_0_Minus_5_Chk =>
27         Result := Subtract (0, -5);
28         Put_Line ("Result: " & Integer'Image (Result));
29     end case;
30 end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

134.2.3 Equality function

Listing 2697: is_equal.ads

```

1 function Is_Equal (A, B : Integer) return Boolean;

```

Listing 2698: is_equal.adb

```

1 function Is_Equal (A, B : Integer) return Boolean is
2 begin
3     return A = B;
4 end Is_Equal;

```

Listing 2699: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Is_Equal;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Equal_Chk,
9          Inequal_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Display_Equal (A, B : Integer;
14                                 Equal : Boolean) is
15             begin

```

(continues on next page)

(continued from previous page)

```

16      Put (Integer'Image (A));
17      if Equal then
18          Put (" is equal to ");
19      else
20          Put (" isn't equal to ");
21      end if;
22      Put_Line (Integer'Image (B) & ".");
23  end Display_Equal;
24
25  Result : Boolean;
26  begin
27      case TC is
28      when Equal_Chk =>
29          for I in 0 .. 10 loop
30              Result := Is_Equal (I, I);
31              Display_Equal (I, I, Result);
32          end loop;
33      when Inequal_Chk =>
34          for I in 0 .. 10 loop
35              Result := Is_Equal (I, I - 1);
36              Display_Equal (I, I - 1, Result);
37          end loop;
38      end case;
39  end Check;
40
41  begin
42      if Argument_Count < 1 then
43          Put_Line ("ERROR: missing arguments! Exiting...");
44          return;
45      elsif Argument_Count > 1 then
46          Put_Line ("Ignoring additional arguments...");
47      end if;
48
49      Check (Test_Case_Index'Value (Argument (1)));
50  end Main;

```

134.2.4 States

Listing 2700: display_state.ads

```

1  procedure Display_State (State : Integer);

```

Listing 2701: display_state.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Display_State (State : Integer) is
4  begin
5      case State is
6      when 0 =>
7          Put_Line ("Off");
8      when 1 =>
9          Put_Line ("On: Simple Processing");
10     when 2 =>
11         Put_Line ("On: Advanced Processing");
12     when others =>
13         null;
14     end case;
15  end Display_State;

```

Listing 2702: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Display_State;
5
6  procedure Main is
7      State : Integer;
8  begin
9      if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Display_State (State);
19 end Main;

```

134.2.5 States #2

Listing 2703: get_state.ads

```

1  function Get_State (State : Integer) return String;

```

Listing 2704: get_state.adb

```

1  function Get_State (State : Integer) return String is
2  begin
3      return (case State is
4          when 0 => "Off",
5          when 1 => "On: Simple Processing",
6          when 2 => "On: Advanced Processing",
7          when others => "");
8  end Get_State;

```

Listing 2705: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Get_State;
5
6  procedure Main is
7      State : Integer;
8  begin
9      if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");
11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Put_Line (Get_State (State));
19 end Main;

```

134.2.6 States #3

Listing 2706: is_on.ads

```
1 function Is_On (State : Integer) return Boolean;
```

Listing 2707: is_on.adb

```
1 function Is_On (State : Integer) return Boolean is
2 begin
3   return not (State = 0);
4 end Is_On;
```

Listing 2708: display_on_off.ads

```
1 procedure Display_On_Off (State : Integer);
```

Listing 2709: display_on_off.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Is_On;
3
4 procedure Display_On_Off (State : Integer) is
5 begin
6   Put_Line (if Is_On (State) then "On" else "Off");
7 end Display_On_Off;
```

Listing 2710: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_On_Off;
5 with Is_On;
6
7 procedure Main is
8   State : Integer;
9 begin
10  if Argument_Count < 1 then
11    Put_Line ("ERROR: missing arguments! Exiting...");
12    return;
13  elsif Argument_Count > 1 then
14    Put_Line ("Ignoring additional arguments...");
15  end if;
16
17  State := Integer'Value (Argument (1));
18
19  Display_On_Off (State);
20  Put_Line (Boolean'Image (Is_On (State)));
21 end Main;
```

134.2.7 States #4

Listing 2711: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 2712: set_next.adb

```

1 procedure Set_Next (State : in out Integer) is
2 begin
3   State := (if State < 2 then State + 1 else 0);
4 end Set_Next;

```

Listing 2713: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");
11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");
14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Set_Next (State);
19   Put_Line (Integer'Image (State));
20 end Main;

```

134.3 Modular Programming

134.3.1 Months

Listing 2714: months.ads

```

1 package Months is
2
3   Jan : constant String := "January";
4   Feb : constant String := "February";
5   Mar : constant String := "March";
6   Apr : constant String := "April";
7   May : constant String := "May";
8   Jun : constant String := "June";
9   Jul : constant String := "July";
10  Aug : constant String := "August";
11  Sep : constant String := "September";
12  Oct : constant String := "October";
13  Nov : constant String := "November";
14  Dec : constant String := "December";
15
16  procedure Display_Months;
17
18 end Months;

```

Listing 2715: months.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Months is
4
5     procedure Display_Months is
6     begin
7         Put_Line ("Months:");
8         Put_Line ("- " & Jan);
9         Put_Line ("- " & Feb);
10        Put_Line ("- " & Mar);
11        Put_Line ("- " & Apr);
12        Put_Line ("- " & May);
13        Put_Line ("- " & Jun);
14        Put_Line ("- " & Jul);
15        Put_Line ("- " & Aug);
16        Put_Line ("- " & Sep);
17        Put_Line ("- " & Oct);
18        Put_Line ("- " & Nov);
19        Put_Line ("- " & Dec);
20    end Display_Months;
21
22 end Months;
```

Listing 2716: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Months;           use Months;
5
6 procedure Main is
7
8     type Test_Case_Index is
9         (Months_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Months_Chk =>
15                 Display_Months;
16         end case;
17     end Check;
18
19 begin
20     if Argument_Count < 1 then
21         Put_Line ("ERROR: missing arguments! Exiting...");
22         return;
23     elsif Argument_Count > 1 then
24         Put_Line ("Ignoring additional arguments...");
25     end if;
26
27     Check (Test_Case_Index'Value (Argument (1)));
28 end Main;
```

134.3.2 Operations

Listing 2717: operations.ads

```

1 package Operations is
2
3     function Add (A, B : Integer) return Integer;
4
5     function Subtract (A, B : Integer) return Integer;
6
7     function Multiply (A, B : Integer) return Integer;
8
9     function Divide (A, B : Integer) return Integer;
10
11 end Operations;
```

Listing 2718: operations.adb

```

1 package body Operations is
2
3     function Add (A, B : Integer) return Integer is
4     begin
5         return A + B;
6     end Add;
7
8     function Subtract (A, B : Integer) return Integer is
9     begin
10        return A - B;
11    end Subtract;
12
13    function Multiply (A, B : Integer) return Integer is
14    begin
15        return A * B;
16    end Multiply;
17
18    function Divide (A, B : Integer) return Integer is
19    begin
20        return A / B;
21    end Divide;
22
23 end Operations;
```

Listing 2719: operations-test.ads

```

1 package Operations.Test is
2
3     procedure Display (A, B : Integer);
4
5 end Operations.Test;
```

Listing 2720: operations-test.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Operations.Test is
4
5     procedure Display (A, B : Integer) is
6         A_Str : constant String := Integer'Image (A);
7         B_Str : constant String := Integer'Image (B);
8     begin
9         Put_Line ("Operations:");
```

(continues on next page)

(continued from previous page)

```

10   Put_Line (A_Str & " + " & B_Str & " = "
11             & Integer'Image (Add (A, B))
12             & ",");
13   Put_Line (A_Str & " - " & B_Str & " = "
14             & Integer'Image (Subtract (A, B))
15             & ",");
16   Put_Line (A_Str & " * " & B_Str & " = "
17             & Integer'Image (Multiply (A, B))
18             & ",");
19   Put_Line (A_Str & " / " & B_Str & " = "
20             & Integer'Image (Divide (A, B))
21             & ",");
22   end Display;
23
24   end Operations.Test;

```

Listing 2721: main.adb

```

1   with Ada.Command_Line; use Ada.Command_Line;
2   with Ada.Text_IO;      use Ada.Text_IO;
3
4   with Operations;
5   with Operations.Test; use Operations.Test;
6
7   procedure Main is
8
9       type Test_Case_Index is
10          (Operations_Chk,
11           Operations_Display_Chk);
12
13       procedure Check (TC : Test_Case_Index) is
14       begin
15           case TC is
16               when Operations_Chk =>
17                   Put_Line ("Add (100, 2) = "
18                             & Integer'Image (Operations.Add (100, 2)));
19                   Put_Line ("Subtract (100, 2) = "
20                             & Integer'Image (Operations.Subtract (100, 2)));
21                   Put_Line ("Multiply (100, 2) = "
22                             & Integer'Image (Operations.Multiply (100, 2)));
23                   Put_Line ("Divide (100, 2) = "
24                             & Integer'Image (Operations.Divide (100, 2)));
25               when Operations_Display_Chk =>
26                   Display (10, 5);
27                   Display (1, 2);
28           end case;
29       end Check;
30
31   begin
32       if Argument_Count < 1 then
33           Put_Line ("ERROR: missing arguments! Exiting...");
34           return;
35       elsif Argument_Count > 1 then
36           Put_Line ("Ignoring additional arguments...");
37       end if;
38
39       Check (Test_Case_Index'Value (Argument (1)));
40   end Main;

```

134.4 Strongly typed language

134.4.1 Colors

Listing 2722: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,
20     Green,
21     Blue);
22
23   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25 end Color_Types;
```

Listing 2723: color_types.adb

```

1 package body Color_Types is
2
3   function To_Integer (C : HTML_Color) return Integer is
4   begin
5     case C is
6       when Salmon    => return 16#FA8072#;
7       when Firebrick => return 16#B22222#;
8       when Red       => return 16#FF0000#;
9       when Darkred   => return 16#8B0000#;
10      when Lime       => return 16#00FF00#;
11      when Forestgreen => return 16#228B22#;
12      when Green      => return 16#008000#;
13      when Darkgreen  => return 16#006400#;
14      when Blue       => return 16#0000FF#;
15      when Mediumblue => return 16#0000CD#;
16      when Darkblue   => return 16#00008B#;
17    end case;
18
19   end To_Integer;
20
21   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
22   begin
23     case C is
24       when Red    => return Red;
25       when Green  => return Green;
26       when Blue   => return Blue;
27    end case;
28
```

(continues on next page)

(continued from previous page)

```

28     end To_HTML_Color;
29
30 end Color_Types;

```

Listing 2724: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3  with Ada.Integer_Text_IO;
4
5  with Color_Types;       use Color_Types;
6
7  procedure Main is
8      type Test_Case_Index is
9          (HTML_Color_Range,
10           HTML_Color_To_Integer,
11           Basic_HTML_Color_To_HTML_Color);
12
13     procedure Check (TC : Test_Case_Index) is
14     begin
15         case TC is
16             when HTML_Color_Range =>
17                 for I in HTML_Color'Range loop
18                     Put_Line (HTML_Color'Image (I));
19                 end loop;
20             when HTML_Color_To_Integer =>
21                 for I in HTML_Color'Range loop
22                     Ada.Integer_Text_IO.Put (Item => To_Integer (I),
23                                             Width => 1,
24                                             Base  => 16);
25                     New_Line;
26                 end loop;
27             when Basic_HTML_Color_To_HTML_Color =>
28                 for I in Basic_HTML_Color'Range loop
29                     Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30                 end loop;
31             end case;
32         end Check;
33
34     begin
35         if Argument_Count < 1 then
36             Put_Line ("ERROR: missing arguments! Exiting...");
37             return;
38         elsif Argument_Count > 1 then
39             Put_Line ("Ignoring additional arguments...");
40         end if;
41
42         Check (Test_Case_Index'Value (Argument (1)));
43     end Main;

```

134.4.2 Integers

Listing 2725: int_types.ads

```

1  package Int_Types is
2
3      type I_100 is range 0 .. 100;
4
5      type U_100 is mod 101;

```

(continues on next page)

(continued from previous page)

```

6  function To_I_100 (V : U_100) return I_100;
7
8  function To_U_100 (V : I_100) return U_100;
9
10 type D_50 is new I_100 range 10 .. 50;
11
12 subtype S_50 is I_100 range 10 .. 50;
13
14 function To_D_50 (V : I_100) return D_50;
15
16 function To_S_50 (V : I_100) return S_50;
17
18 function To_I_100 (V : D_50) return I_100;
19
20
21 end Int_Types;

```

Listing 2726: int_types.adb

```

1  package body Int_Types is
2
3      function To_I_100 (V : U_100) return I_100 is
4      begin
5          return I_100 (V);
6      end To_I_100;
7
8      function To_U_100 (V : I_100) return U_100 is
9      begin
10         return U_100 (V);
11     end To_U_100;
12
13     function To_D_50 (V : I_100) return D_50 is
14     Min : constant I_100 := I_100 (D_50'First);
15     Max : constant I_100 := I_100 (D_50'Last);
16     begin
17         if V > Max then
18             return D_50'Last;
19         elsif V < Min then
20             return D_50'First;
21         else
22             return D_50 (V);
23         end if;
24     end To_D_50;
25
26     function To_S_50 (V : I_100) return S_50 is
27     begin
28         if V > S_50'Last then
29             return S_50'Last;
30         elsif V < S_50'First then
31             return S_50'First;
32         else
33             return V;
34         end if;
35     end To_S_50;
36
37     function To_I_100 (V : D_50) return I_100 is
38     begin
39         return I_100 (V);
40     end To_I_100;
41
42 end Int_Types;

```

Listing 2727: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7      package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8      package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9      package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11     use I_100_IO;
12     use U_100_IO;
13     use D_50_IO;
14
15     type Test_Case_Index is
16         (I_100_Range,
17          U_100_Range,
18          U_100_Wraparound,
19          U_100_To_I_100,
20          I_100_To_U_100,
21          D_50_Range,
22          S_50_Range,
23          I_100_To_D_50,
24          I_100_To_S_50,
25          D_50_To_I_100,
26          S_50_To_I_100);
27
28     procedure Check (TC : Test_Case_Index) is
29     begin
30         I_100_IO.Default_Width := 1;
31         U_100_IO.Default_Width := 1;
32         D_50_IO.Default_Width  := 1;
33
34         case TC is
35             when I_100_Range =>
36                 Put (I_100'First);
37                 New_Line;
38                 Put (I_100'Last);
39                 New_Line;
40             when U_100_Range =>
41                 Put (U_100'First);
42                 New_Line;
43                 Put (U_100'Last);
44                 New_Line;
45             when U_100_Wraparound =>
46                 Put (U_100'First - 1);
47                 New_Line;
48                 Put (U_100'Last + 1);
49                 New_Line;
50             when U_100_To_I_100 =>
51                 for I in U_100'Range loop
52                     I_100_IO.Put (To_I_100 (I));
53                     New_Line;
54                 end loop;
55             when I_100_To_U_100 =>
56                 for I in I_100'Range loop
57                     Put (To_U_100 (I));
58                     New_Line;
59                 end loop;

```

(continues on next page)

(continued from previous page)

```

60     when D_50_Range =>
61         Put (D_50'First);
62         New_Line;
63         Put (D_50'Last);
64         New_Line;
65     when S_50_Range =>
66         Put (S_50'First);
67         New_Line;
68         Put (S_50'Last);
69         New_Line;
70     when I_100_To_D_50 =>
71         for I in I_100'Range loop
72             Put (To_D_50 (I));
73             New_Line;
74         end loop;
75     when I_100_To_S_50 =>
76         for I in I_100'Range loop
77             Put (To_S_50 (I));
78             New_Line;
79         end loop;
80     when D_50_To_I_100 =>
81         for I in D_50'Range loop
82             Put (To_I_100 (I));
83             New_Line;
84         end loop;
85     when S_50_To_I_100 =>
86         for I in S_50'Range loop
87             Put (I);
88             New_Line;
89         end loop;
90     end case;
91 end Check;
92
93 begin
94     if Argument_Count < 1 then
95         Put_Line ("ERROR: missing arguments! Exiting...");
96         return;
97     elsif Argument_Count > 1 then
98         Put_Line ("Ignoring additional arguments...");
99     end if;
100
101     Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

134.4.3 Temperatures

Listing 2728: temperature_types.ads

```

1 package Temperature_Types is
2
3     type Celsius is digits 6 range -273.15 .. 5504.85;
4
5     type Int_Celsius is range -273 .. 5505;
6
7     function To_Celsius (T : Int_Celsius) return Celsius;
8
9     function To_Int_Celsius (T : Celsius) return Int_Celsius;
10
11     type Kelvin is digits 6 range 0.0 .. 5778.00;
12

```

(continues on next page)

(continued from previous page)

```

13  function To_Celsius (T : Kelvin) return Celsius;
14
15  function To_Kelvin (T : Celsius) return Kelvin;
16
17  end Temperature_Types;

```

Listing 2729: temperature_types.adb

```

1  package body Temperature_Types is
2
3      function To_Celsius (T : Int_Celsius) return Celsius is
4          Min : constant Float := Float (Celsius'First);
5          Max : constant Float := Float (Celsius'Last);
6
7          F   : constant Float := Float (T);
8      begin
9          if F > Max then
10             return Celsius (Max);
11          elsif F < Min then
12             return Celsius (Min);
13          else
14             return Celsius (F);
15          end if;
16      end To_Celsius;
17
18      function To_Int_Celsius (T : Celsius) return Int_Celsius is
19      begin
20          return Int_Celsius (T);
21      end To_Int_Celsius;
22
23      function To_Celsius (T : Kelvin) return Celsius is
24          F : constant Float := Float (T);
25      begin
26          return Celsius (F - 273.15);
27      end To_Celsius;
28
29      function To_Kelvin (T : Celsius) return Kelvin is
30          F : constant Float := Float (T);
31      begin
32          return Kelvin (F + 273.15);
33      end To_Kelvin;
34
35  end Temperature_Types;

```

Listing 2730: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Temperature_Types; use Temperature_Types;
5
6  procedure Main is
7      package Celsius_IO   is new Ada.Text_IO.Float_IO (Celsius);
8      package Kelvin_IO    is new Ada.Text_IO.Float_IO (Kelvin);
9      package Int_Celsius_IO is new Ada.Text_IO.Integer_IO (Int_Celsius);
10
11      use Celsius_IO;
12      use Kelvin_IO;
13      use Int_Celsius_IO;
14

```

(continues on next page)

(continued from previous page)

```

15  type Test_Case_Index is
16      (Celsius_Range,
17       Celsius_To_Int_Celsius,
18       Int_Celsius_To_Celsius,
19       Kelvin_To_Celsius,
20       Celsius_To_Kelvin);
21
22  procedure Check (TC : Test_Case_Index) is
23  begin
24      Celsius_IO.Default_Fore := 1;
25      Kelvin_IO.Default_Fore  := 1;
26      Int_Celsius_IO.Default_Width := 1;
27
28      case TC is
29          when Celsius_Range =>
30              Put (Celsius'First);
31              New_Line;
32              Put (Celsius'Last);
33              New_Line;
34          when Celsius_To_Int_Celsius =>
35              Put (To_Int_Celsius (Celsius'First));
36              New_Line;
37              Put (To_Int_Celsius (0.0));
38              New_Line;
39              Put (To_Int_Celsius (Celsius'Last));
40              New_Line;
41          when Int_Celsius_To_Celsius =>
42              Put (To_Celsius (Int_Celsius'First));
43              New_Line;
44              Put (To_Celsius (0));
45              New_Line;
46              Put (To_Celsius (Int_Celsius'Last));
47              New_Line;
48          when Kelvin_To_Celsius =>
49              Put (To_Celsius (Kelvin'First));
50              New_Line;
51              Put (To_Celsius (0));
52              New_Line;
53              Put (To_Celsius (Kelvin'Last));
54              New_Line;
55          when Celsius_To_Kelvin =>
56              Put (To_Kelvin (Celsius'First));
57              New_Line;
58              Put (To_Kelvin (Celsius'Last));
59              New_Line;
60      end case;
61  end Check;
62
63  begin
64      if Argument_Count < 1 then
65          Put_Line ("ERROR: missing arguments! Exiting...");
66          return;
67      elsif Argument_Count > 1 then
68          Put_Line ("Ignoring additional arguments...");
69      end if;
70
71      Check (Test_Case_Index'Value (Argument (1)));
72  end Main;

```

134.5 Records

134.5.1 Directions

Listing 2731: directions.ads

```

1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northeast,
8      East,
9      Southeast,
10     South,
11     Southwest,
12     West,
13     Northwest);
14
15   function To_Direction (N: Angle_Mod) return Direction;
16
17   type Ext_Angle is record
18     Angle_Elem      : Angle_Mod;
19     Direction_Elem  : Direction;
20   end record;
21
22   function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
23
24   procedure Display (N : Ext_Angle);
25
26 end Directions;
```

Listing 2732: directions.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Directions is
4
5   procedure Display (N : Ext_Angle) is
6   begin
7     Put_Line ("Angle: "
8              & Angle_Mod'Image (N.Angle_Elem)
9              & " => "
10             & Direction'Image (N.Direction_Elem)
11             & ".");
12   end Display;
13
14   function To_Direction (N : Angle_Mod) return Direction is
15   begin
16     case N is
17       when 0      => return North;
18       when 1 .. 89 => return Northeast;
19       when 90     => return East;
20       when 91 .. 179 => return Southeast;
21       when 180    => return South;
22       when 181 .. 269 => return Southwest;
23       when 270    => return West;
24       when 271 .. 359 => return Northwest;
25     end case;
26   end To_Direction;
```

(continues on next page)

(continued from previous page)

```

27
28     function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29     begin
30         return (Angle_Elem    => N,
31                Direction_Elem => To_Direction (N));
32     end To_Ext_Angle;
33
34 end Directions;

```

Listing 2733: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Directions;       use Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Direction_Chk =>
14             Display (To_Ext_Angle (0));
15             Display (To_Ext_Angle (30));
16             Display (To_Ext_Angle (45));
17             Display (To_Ext_Angle (90));
18             Display (To_Ext_Angle (91));
19             Display (To_Ext_Angle (120));
20             Display (To_Ext_Angle (180));
21             Display (To_Ext_Angle (250));
22             Display (To_Ext_Angle (270));
23         end case;
24     end Check;
25
26     begin
27         if Argument_Count < 1 then
28             Put_Line ("ERROR: missing arguments! Exiting...");
29             return;
30         elsif Argument_Count > 1 then
31             Put_Line ("Ignoring additional arguments...");
32         end if;
33
34         Check (Test_Case_Index'Value (Argument (1)));
35     end Main;

```

134.5.2 Colors

Listing 2734: color_types.ads

```

1  package Color_Types is
2
3      type HTML_Color is
4          (Salmon,
5           Firebrick,
6           Red,
7           Darkred,
8           Lime,

```

(continues on next page)

(continued from previous page)

```

9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16     function To_Integer (C : HTML_Color) return Integer;
17
18     type Basic_HTML_Color is
19       (Red,
20        Green,
21        Blue);
22
23     function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25     subtype Int_Color is Integer range 0 .. 255;
26
27     type RGB is record
28       Red   : Int_Color;
29       Green : Int_Color;
30       Blue  : Int_Color;
31     end record;
32
33     function To_RGB (C : HTML_Color) return RGB;
34
35     function Image (C : RGB) return String;
36
37 end Color_Types;

```

Listing 2735: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_Integer (C : HTML_Color) return Integer is
6      begin
7          case C is
8              when Salmon    => return 16#FA8072#;
9              when Firebrick => return 16#B22222#;
10             when Red        => return 16#FF0000#;
11             when Darkred    => return 16#8B0000#;
12             when Lime       => return 16#00FF00#;
13             when Forestgreen => return 16#228B22#;
14             when Green      => return 16#008000#;
15             when Darkgreen  => return 16#006400#;
16             when Blue       => return 16#0000FF#;
17             when Mediumblue => return 16#0000CD#;
18             when Darkblue   => return 16#00008B#;
19          end case;
20
21      end To_Integer;
22
23      function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24      begin
25          case C is
26              when Red    => return Red;
27              when Green  => return Green;
28              when Blue   => return Blue;
29          end case;

```

(continues on next page)

(continued from previous page)

```

30 end To_HTML_Color;
31
32 function To_RGB (C : HTML_Color) return RGB is
33 begin
34     case C is
35         when Salmon      => return (16#FA#, 16#80#, 16#72#);
36         when Firebrick   => return (16#B2#, 16#22#, 16#22#);
37         when Red         => return (16#FF#, 16#00#, 16#00#);
38         when Darkred     => return (16#8B#, 16#00#, 16#00#);
39         when Lime        => return (16#00#, 16#FF#, 16#00#);
40         when Forestgreen => return (16#22#, 16#8B#, 16#22#);
41         when Green       => return (16#00#, 16#80#, 16#00#);
42         when Darkgreen   => return (16#00#, 16#64#, 16#00#);
43         when Blue        => return (16#00#, 16#00#, 16#FF#);
44         when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
45         when Darkblue    => return (16#00#, 16#00#, 16#8B#);
46     end case;
47
48 end To_RGB;
49
50 function Image (C : RGB) return String is
51     subtype Str_Range is Integer range 1 .. 10;
52     SR : String (Str_Range);
53     SG : String (Str_Range);
54     SB : String (Str_Range);
55 begin
56     Ada.Integer_Text_IO.Put (To  => SR,
57                             Item => C.Red,
58                             Base => 16);
59     Ada.Integer_Text_IO.Put (To  => SG,
60                             Item => C.Green,
61                             Base => 16);
62     Ada.Integer_Text_IO.Put (To  => SB,
63                             Item => C.Blue,
64                             Base => 16);
65     return ("(Red => " & SR
66           & ", Green => " & SG
67           & ", Blue => " & SB
68           & ")");
69 end Image;
70
71 end Color_Types;

```

Listing 2736: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Color_Types;      use Color_Types;
5
6 procedure Main is
7     type Test_Case_Index is
8         (HTML_Color_To_RGB);
9
10    procedure Check (TC : Test_Case_Index) is
11    begin
12        case TC is
13            when HTML_Color_To_RGB =>
14                for I in HTML_Color'Range loop
15                    Put_Line (HTML_Color'Image (I) & " => "
16                            & Image (To_RGB (I)) & ".");

```

(continues on next page)

(continued from previous page)

```

17         end loop;
18     end case;
19 end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

134.5.3 Inventory

Listing 2737: inventory_pkg.ads

```

1 package Inventory_Pkg is
2
3     type Item_Name is
4         (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6     function To_String (I : Item_Name) return String;
7
8     type Item is record
9         Name      : Item_Name;
10        Quantity  : Natural;
11        Price     : Float;
12    end record;
13
14    function Init (Name      : Item_Name;
15                  Quantity  : Natural;
16                  Price     : Float) return Item;
17
18    procedure Add (Assets : in out Float;
19                  I       : Item);
20
21 end Inventory_Pkg;

```

Listing 2738: inventory_pkg.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inventory_Pkg is
4
5     function To_String (I : Item_Name) return String is
6     begin
7         case I is
8             when Ballpoint_Pen      => return "Ballpoint Pen";
9             when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10            when Feather_Quill_Pen    => return "Feather Quill Pen";
11        end case;
12    end To_String;
13
14    function Init (Name      : Item_Name;
15                  Quantity  : Natural;
16                  Price     : Float) return Item is

```

(continues on next page)

(continued from previous page)

```

17  begin
18      Put_Line ("Item: " & To_String (Name) & ".");
19
20      return (Name      => Name,
21              Quantity => Quantity,
22              Price     => Price);
23  end Init;
24
25  procedure Add (Assets : in out Float;
26                I       : Item) is
27  begin
28      Assets := Assets + Float (I.Quantity) * I.Price;
29  end Add;
30
31  end Inventory_Pkg;

```

Listing 2739: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Inventory_Pkg;    use Inventory_Pkg;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42);
9
10     type Test_Case_Index is
11         (Inventory_Chk);
12
13     procedure Display (Assets : Float) is
14         package F_IO is new Ada.Text_IO.Float_IO (Float);
15
16         use F_IO;
17     begin
18         Put ("Assets: $");
19         Put (Assets, 1, 2, 0);
20         Put (".");
21         New_Line;
22     end Display;
23
24     procedure Check (TC : Test_Case_Index) is
25         I : Item;
26         Assets : Float := 0.0;
27
28         -- Please ignore the following three lines!
29         pragma Warnings (Off, "default initialization");
30         for Assets'Address use F'Address;
31         pragma Warnings (On, "default initialization");
32     begin
33         case TC is
34         when Inventory_Chk =>
35             I := Init (Ballpoint_Pen,      185,  0.15);
36             Add (Assets, I);
37             Display (Assets);
38
39             I := Init (Oil_Based_Pen_Marker, 100,  9.0);
40             Add (Assets, I);
41             Display (Assets);
42
43             I := Init (Feather_Quill_Pen,   2, 40.0);

```

(continues on next page)

(continued from previous page)

```

44     Add (Assets, I);
45     Display (Assets);
46   end case;
47 end Check;
48
49 begin
50   if Argument_Count < 1 then
51     Put_Line ("ERROR: missing arguments! Exiting...");
52     return;
53   elsif Argument_Count > 1 then
54     Put_Line ("Ignoring additional arguments...");
55   end if;
56
57   Check (Test_Case_Index'Value (Argument (1)));
58 end Main;

```

134.6 Arrays

134.6.1 Constrained Array

Listing 2740: constrained_arrays.ads

```

1 package Constrained_Arrays is
2
3   type My_Index is range 1 .. 10;
4
5   type My_Array is array (My_Index) of Integer;
6
7   function Init return My_Array;
8
9   procedure Double (A : in out My_Array);
10
11  function First_Elem (A : My_Array) return Integer;
12
13  function Last_Elem (A : My_Array) return Integer;
14
15  function Length (A : My_Array) return Integer;
16
17  A : My_Array := (1, 2, others => 42);
18
19 end Constrained_Arrays;

```

Listing 2741: constrained_arrays.adb

```

1 package body Constrained_Arrays is
2
3   function Init return My_Array is
4     A : My_Array;
5   begin
6     for I in My_Array'Range loop
7       A (I) := Integer (I);
8     end loop;
9
10    return A;
11  end Init;
12
13  procedure Double (A : in out My_Array) is
14  begin

```

(continues on next page)

(continued from previous page)

```

15     for I in A'Range loop
16         A (I) := A (I) * 2;
17     end loop;
18 end Double;
19
20 function First_Elem (A : My_Array) return Integer is
21 begin
22     return A (A'First);
23 end First_Elem;
24
25 function Last_Elem (A : My_Array) return Integer is
26 begin
27     return A (A'Last);
28 end Last_Elem;
29
30 function Length (A : My_Array) return Integer is
31 begin
32     return A'Length;
33 end Length;
34
35 end Constrained_Arrays;

```

Listing 2742: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Constrained_Arrays; use Constrained_Arrays;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Range_Chk,
9           Array_Range_Chk,
10          A_Obj_Chk,
11          Init_Chk,
12          Double_Chk,
13          First_Elem_Chk,
14          Last_Elem_Chk,
15          Length_Chk);
16
17  procedure Check (TC : Test_Case_Index) is
18      AA : My_Array;
19
20      procedure Display (A : My_Array) is
21      begin
22          for I in A'Range loop
23              Put_Line (Integer'Image (A (I)));
24          end loop;
25      end Display;
26
27      procedure Local_Init (A : in out My_Array) is
28      begin
29          A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30      end Local_Init;
31  begin
32      case TC is
33      when Range_Chk =>
34          for I in My_Index loop
35              Put_Line (My_Index'Image (I));
36          end loop;
37      when Array_Range_Chk =>

```

(continues on next page)

(continued from previous page)

```

38     for I in My_Array'Range loop
39         Put_Line (My_Index'Image (I));
40     end loop;
41     when A_Obj_Chk =>
42         Display (A);
43     when Init_Chk =>
44         AA := Init;
45         Display (AA);
46     when Double_Chk =>
47         Local_Init (AA);
48         Double (AA);
49         Display (AA);
50     when First_Elem_Chk =>
51         Local_Init (AA);
52         Put_Line (Integer'Image (First_Elem (AA)));
53     when Last_Elem_Chk =>
54         Local_Init (AA);
55         Put_Line (Integer'Image (Last_Elem (AA)));
56     when Length_Chk =>
57         Put_Line (Integer'Image (Length (AA)));
58     end case;
59 end Check;
60
61 begin
62     if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");
64         return;
65     elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");
67     end if;
68
69     Check (Test_Case_Index'Value (Argument (1)));
70 end Main;

```

134.6.2 Colors: Lookup-Table

Listing 2743: color_types.ads

```

1  package Color_Types is
2
3      type HTML_Color is
4          (Salmon,
5           Firebrick,
6           Red,
7           Darkred,
8           Lime,
9           Forestgreen,
10          Green,
11          Darkgreen,
12          Blue,
13          Mediumblue,
14          Darkblue);
15
16      subtype Int_Color is Integer range 0 .. 255;
17
18      type RGB is record
19          Red   : Int_Color;
20          Green : Int_Color;
21          Blue  : Int_Color;
22      end record;

```

(continues on next page)

(continued from previous page)

```

23
24 function To_RGB (C : HTML_Color) return RGB;
25
26 function Image (C : RGB) return String;
27
28 type HTML_Color_RGB is array (HTML_Color) of RGB;
29
30 To_RGB_Lookup_Table : constant HTML_Color_RGB
31 := (Salmon    => (16#FA#, 16#80#, 16#72#),
32      Firebrick => (16#B2#, 16#22#, 16#22#),
33      Red       => (16#FF#, 16#00#, 16#00#),
34      Darkred   => (16#8B#, 16#00#, 16#00#),
35      Lime      => (16#00#, 16#FF#, 16#00#),
36      Forestgreen => (16#22#, 16#8B#, 16#22#),
37      Green     => (16#00#, 16#80#, 16#00#),
38      Darkgreen => (16#00#, 16#64#, 16#00#),
39      Blue      => (16#00#, 16#00#, 16#FF#),
40      Mediumblue => (16#00#, 16#00#, 16#CD#),
41      Darkblue  => (16#00#, 16#00#, 16#8B#));
42
43 end Color_Types;

```

Listing 2744: color_types.adb

```

1 with Ada.Integer_Text_IO;
2 package body Color_Types is
3
4     function To_RGB (C : HTML_Color) return RGB is
5     begin
6         return To_RGB_Lookup_Table (C);
7     end To_RGB;
8
9     function Image (C : RGB) return String is
10     subtype Str_Range is Integer range 1 .. 10;
11     SR : String (Str_Range);
12     SG : String (Str_Range);
13     SB : String (Str_Range);
14     begin
15         Ada.Integer_Text_IO.Put (To    => SR,
16                                Item   => C.Red,
17                                Base   => 16);
18         Ada.Integer_Text_IO.Put (To    => SG,
19                                Item   => C.Green,
20                                Base   => 16);
21         Ada.Integer_Text_IO.Put (To    => SB,
22                                Item   => C.Blue,
23                                Base   => 16);
24
25         return ("(Red => " & SR
26                & ", Green => " & SG
27                & ", Blue => " & SB
28                & ")");
29     end Image;
30 end Color_Types;

```

Listing 2745: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3

```

(continues on next page)

(continued from previous page)

```

4  with Color_Types;           use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Color_Table_Chk,
9           HTML_Color_To_Integer_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12     begin
13         case TC is
14             when Color_Table_Chk =>
15                 Put_Line ("Size of HTML_Color_RGB: "
16                     & Integer'Image (HTML_Color_RGB'Length));
17                 Put_Line ("Firebrick: "
18                     & Image (To_RGB_Lookup_Table (Firebrick)));
19             when HTML_Color_To_Integer_Chk =>
20                 for I in HTML_Color'Range loop
21                     Put_Line (HTML_Color'Image (I) & " => "
22                         & Image (To_RGB (I)) & ".");
23                 end loop;
24             end case;
25     end Check;
26
27 begin
28     if Argument_Count < 1 then
29         Put_Line ("ERROR: missing arguments! Exiting...");
30         return;
31     elsif Argument_Count > 1 then
32         Put_Line ("Ignoring additional arguments...");
33     end if;
34
35     Check (Test_Case_Index'Value (Argument (1)));
36 end Main;

```

134.6.3 Unconstrained Array

Listing 2746: unconstrained_arrays.ads

```

1  package Unconstrained_Arrays is
2
3      type My_Array is array (Positive range <>) of Integer;
4
5      procedure Init (A : in out My_Array);
6
7      function Init (I, L : Positive) return My_Array;
8
9      procedure Double (A : in out My_Array);
10
11     function Diff_Prev_Elem (A : My_Array) return My_Array;
12
13 end Unconstrained_Arrays;

```

Listing 2747: unconstrained_arrays.adb

```

1  package body Unconstrained_Arrays is
2
3      procedure Init (A : in out My_Array) is
4          Y : Natural := A'Last;
5      begin

```

(continues on next page)

(continued from previous page)

```

6      for I in A'Range loop
7          A (I) := Y;
8          Y := Y - 1;
9      end loop;
10     end Init;
11
12     function Init (I, L : Positive) return My_Array is
13         A : My_Array (I .. I + L - 1);
14     begin
15         Init (A);
16         return A;
17     end Init;
18
19     procedure Double (A : in out My_Array) is
20     begin
21         for I in A'Range loop
22             A (I) := A (I) * 2;
23         end loop;
24     end Double;
25
26     function Diff_Prev_Elem (A : My_Array) return My_Array is
27         A_Out : My_Array (A'Range);
28     begin
29         A_Out (A'First) := 0;
30         for I in A'First + 1 .. A'Last loop
31             A_Out (I) := A (I) - A (I - 1);
32         end loop;
33
34         return A_Out;
35     end Diff_Prev_Elem;
36
37 end Unconstrained_Arrays;

```

Listing 2748: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Init_Chk,
9           Init_Proc_Chk,
10          Double_Chk,
11          Diff_Prev_Chk,
12          Diff_Prev_Single_Chk);
13
14      procedure Check (TC : Test_Case_Index) is
15          AA : My_Array (1 .. 5);
16          AB : My_Array (5 .. 9);
17
18          procedure Display (A : My_Array) is
19          begin
20              for I in A'Range loop
21                  Put_Line (Integer'Image (A (I)));
22              end loop;
23          end Display;
24
25          procedure Local_Init (A : in out My_Array) is
26          begin

```

(continues on next page)

(continued from previous page)

```

27     A := (1, 2, 5, 10, -10);
28     end Local_Init;
29
30     begin
31         case TC is
32             when Init_Chk =>
33                 AA := Init (AA'First, AA'Length);
34                 AB := Init (AB'First, AB'Length);
35                 Display (AA);
36                 Display (AB);
37             when Init_Proc_Chk =>
38                 Init (AA);
39                 Init (AB);
40                 Display (AA);
41                 Display (AB);
42             when Double_Chk =>
43                 Local_Init (AB);
44                 Double (AB);
45                 Display (AB);
46             when Diff_Prev_Chk =>
47                 Local_Init (AB);
48                 AB := Diff_Prev_Elem (AB);
49                 Display (AB);
50             when Diff_Prev_Single_Chk =>
51                 declare
52                     A1 : My_Array (1 .. 1) := (1 => 42);
53                 begin
54                     A1 := Diff_Prev_Elem (A1);
55                     Display (A1);
56                 end;
57             end case;
58         end Check;
59
60     begin
61         if Argument_Count < 1 then
62             Put_Line ("ERROR: missing arguments! Exiting...");
63             return;
64         elsif Argument_Count > 1 then
65             Put_Line ("Ignoring additional arguments...");
66         end if;
67
68         Check (Test_Case_Index'Value (Argument (1)));
69     end Main;

```

134.6.4 Product info

Listing 2749: product_info_pkg.ads

```

1  package Product_Info_Pkg is
2
3      subtype Quantity is Natural;
4
5      subtype Currency is Float;
6
7      type Product_Info is record
8          Units : Quantity;
9          Price : Currency;
10     end record;
11
12     type Product_Infos is array (Positive range <>) of Product_Info;

```

(continues on next page)

(continued from previous page)

```

13
14  type Currency_Array is array (Positive range <>) of Currency;
15
16  procedure Total (P : Product_Infos;
17                  Tot : out Currency_Array);
18
19  function Total (P : Product_Infos) return Currency_Array;
20
21  function Total (P : Product_Infos) return Currency;
22
23  end Product_Info_Pkg;

```

Listing 2750: product_info_pkg.adb

```

1  package body Product_Info_Pkg is
2
3      -- Get total for single product
4      function Total (P : Product_Info) return Currency is
5          (Currency (P.Units) * P.Price);
6
7      procedure Total (P : Product_Infos;
8                      Tot : out Currency_Array) is
9      begin
10         for I in P'Range loop
11             Tot (I) := Total (P (I));
12         end loop;
13     end Total;
14
15     function Total (P : Product_Infos) return Currency_Array
16     is
17         Tot : Currency_Array (P'Range);
18     begin
19         Total (P, Tot);
20         return Tot;
21     end Total;
22
23     function Total (P : Product_Infos) return Currency
24     is
25         Tot : Currency := 0.0;
26     begin
27         for I in P'Range loop
28             Tot := Tot + Total (P (I));
29         end loop;
30         return Tot;
31     end Total;
32
33  end Product_Info_Pkg;

```

Listing 2751: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Product_Info_Pkg;   use Product_Info_Pkg;
5
6  procedure Main is
7      package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9      type Test_Case_Index is
10         (Total_Func_Chk,

```

(continues on next page)

(continued from previous page)

```

11     Total_Proc_Chk,
12     Total_Value_Chk);
13
14 procedure Check (TC : Test_Case_Index) is
15     subtype Test_Range is Positive range 1 .. 5;
16
17     P    : Product_Infos (Test_Range);
18     Tots : Currency_Array (Test_Range);
19     Tot  : Currency;
20
21     procedure Display (Tots : Currency_Array) is
22     begin
23         for I in Tots'Range loop
24             Currency_IO.Put (Tots (I));
25             New_Line;
26         end loop;
27     end Display;
28
29     procedure Local_Init (P : in out Product_Infos) is
30     begin
31         P := ((1, 0.5),
32              (2, 10.0),
33              (5, 40.0),
34              (10, 10.0),
35              (10, 20.0));
36     end Local_Init;
37
38 begin
39     Currency_IO.Default_Fore := 1;
40     Currency_IO.Default_Aft  := 2;
41     Currency_IO.Default_Exp  := 0;
42
43     case TC is
44     when Total_Func_Chk =>
45         Local_Init (P);
46         Tots := Total (P);
47         Display (Tots);
48     when Total_Proc_Chk =>
49         Local_Init (P);
50         Total (P, Tots);
51         Display (Tots);
52     when Total_Value_Chk =>
53         Local_Init (P);
54         Tot := Total (P);
55         Currency_IO.Put (Tot);
56         New_Line;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

134.6.5 String_10

Listing 2752: strings_10.ads

```

1 package Strings_10 is
2
3     subtype String_10 is String (1 .. 10);
4
5     -- Using "type String_10 is..." is possible, too.
6
7     function To_String_10 (S : String) return String_10;
8
9 end Strings_10;
```

Listing 2753: strings_10.adb

```

1 package body Strings_10 is
2
3     function To_String_10 (S : String) return String_10 is
4         S_Out : String_10;
5     begin
6         for I in String_10'First .. Integer'Min (String_10'Last, S'Last) loop
7             S_Out (I) := S (I);
8         end loop;
9
10        for I in Integer'Min (String_10'Last + 1, S'Last + 1) .. String_10'Last loop
11            S_Out (I) := ' ';
12        end loop;
13
14        return S_Out;
15    end To_String_10;
16
17 end Strings_10;
```

Listing 2754: main.adb

```

1 with Ada.Command_Line;   use Ada.Command_Line;
2 with Ada.Text_IO;        use Ada.Text_IO;
3
4 with Strings_10;          use Strings_10;
5
6 procedure Main is
7     type Test_Case_Index is
8         (String_10_Long_Chk,
9          String_10_Short_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         SL : constant String := "And this is a long string just for testing...";
13         SS : constant String := "Hey!";
14         S_10 : String_10;
15
16     begin
17         case TC is
18             when String_10_Long_Chk =>
19                 S_10 := To_String_10 (SL);
20                 Put_Line (String (S_10));
21             when String_10_Short_Chk =>
22                 S_10 := (others => ' ');
23                 S_10 := To_String_10 (SS);
24                 Put_Line (String (S_10));
25             end case;
26     end Check;
```

(continues on next page)

(continued from previous page)

```

27
28 begin
29   if Argument_Count < 1 then
30     Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
31     return;
32   elsif Argument_Count > 1 then
33     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
34   end if;
35
36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;

```

134.6.6 List of Names

Listing 2755: names_ages.ads

```

1 package Names_Ages is
2
3   Max_People : constant Positive := 10;
4
5   subtype Name_Type is String (1 .. 50);
6
7   type Age_Type is new Natural;
8
9   type Person is record
10     Name : Name_Type;
11     Age  : Age_Type;
12   end record;
13
14   type People_Array is array (Positive range <>) of Person;
15
16   type People is record
17     People_A : People_Array (1 .. Max_People);
18     Last_Valid : Natural;
19   end record;
20
21   procedure Reset (P : in out People);
22
23   procedure Add (P : in out People;
24                 Name : String);
25
26   function Get (P : People;
27                Name : String) return Age_Type;
28
29   procedure Update (P : in out People;
30                    Name : String;
31                    Age : Age_Type);
32
33   procedure Display (P : People);
34
35 end Names_Ages;

```

Listing 2756: names_ages.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Ada.Strings;      use Ada.Strings;
3 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
4
5 package body Names_Ages is

```

(continues on next page)

(continued from previous page)

```

6
7  function To_Name_Type (S : String) return Name_Type is
8      S_Out : Name_Type := (others => ' ');
9  begin
10     for I in 1 .. Integer'Min (S'Last, Name_Type'Last) loop
11         S_Out (I) := S (I);
12     end loop;
13
14     return S_Out;
15 end To_Name_Type;
16
17 procedure Init (P      : in out Person;
18                Name   :      String) is
19 begin
20     P.Name := To_Name_Type (Name);
21     P.Age := 0;
22 end Init;
23
24 function Match (P      : Person;
25                Name   : String) return Boolean is
26 begin
27     return P.Name = To_Name_Type (Name);
28 end Match;
29
30 function Get (P : Person) return Age_Type is
31 begin
32     return P.Age;
33 end Get;
34
35 procedure Update (P      : in out Person;
36                  Age   :      Age_Type) is
37 begin
38     P.Age := Age;
39 end Update;
40
41 procedure Display (P : Person) is
42 begin
43     Put_Line ("NAME: " & Trim (P.Name, Right));
44     Put_Line ("AGE: " & Age_Type'Image (P.Age));
45 end Display;
46
47 procedure Reset (P : in out People) is
48 begin
49     P.Last_Valid := 0;
50 end Reset;
51
52 procedure Add (P      : in out People;
53               Name   :      String) is
54 begin
55     P.Last_Valid := P.Last_Valid + 1;
56     Init (P.People_A (P.Last_Valid), Name);
57 end Add;
58
59 function Get (P      : People;
60               Name   : String) return Age_Type is
61 begin
62     for I in P.People_A'First .. P.Last_Valid loop
63         if Match (P.People_A (I), Name) then
64             return Get (P.People_A (I));
65         end if;
66     end loop;

```

(continues on next page)

(continued from previous page)

```

67
68     return 0;
69 end Get;
70
71 procedure Update (P      : in out People;
72                  Name    :      String;
73                  Age     :      Age_Type) is
74 begin
75     for I in P.People_A'First .. P.Last_Valid loop
76         if Match (P.People_A (I), Name) then
77             Update (P.People_A (I), Age);
78         end if;
79     end loop;
80 end Update;
81
82 procedure Display (P : People) is
83 begin
84     Put_Line ("LIST OF NAMES:");
85     for I in P.People_A'First .. P.Last_Valid loop
86         Display (P.People_A (I));
87     end loop;
88 end Display;
89
90 end Names_Ages;

```

Listing 2757: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Names_Ages;          use Names_Ages;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Names_Ages_Chk,
9           Get_Age_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         P : People;
13     begin
14         case TC is
15             when Names_Ages_Chk =>
16                 Reset (P);
17                 Add (P, "John");
18                 Add (P, "Patricia");
19                 Add (P, "Josh");
20                 Display (P);
21                 Update (P, "John", 18);
22                 Update (P, "Patricia", 35);
23                 Update (P, "Josh", 53);
24                 Display (P);
25             when Get_Age_Chk =>
26                 Reset (P);
27                 Add (P, "Peter");
28                 Update (P, "Peter", 45);
29                 Put_Line ("Peter is "
30                           & Age_Type'Image (Get (P, "Peter"))
31                           & " years old.");
32         end case;
33     end Check;
34

```

(continues on next page)

(continued from previous page)

```

35 begin
36   if Argument_Count < 1 then
37     Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
38     return;
39   elsif Argument_Count > 1 then
40     Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
41   end if;
42
43   Check (Test_Case_Index'Value (Argument (1)));
44 end Main;

```

134.7 More About Types

134.7.1 Aggregate Initialization

Listing 2758: aggregates.ads

```

1  package Aggregates is
2
3     type Rec is record
4       W : Integer := 10;
5       X : Integer := 11;
6       Y : Integer := 12;
7       Z : Integer := 13;
8     end record;
9
10    type Int_Arr is array (1 .. 20) of Integer;
11
12    procedure Init (R : out Rec);
13
14    procedure Init_Some (A : out Int_Arr);
15
16    procedure Init (A : out Int_Arr);
17
18 end Aggregates;

```

Listing 2759: aggregates.adb

```

1  package body Aggregates is
2
3     procedure Init (R : out Rec) is
4     begin
5       R := (X      => 100,
6             Y      => 200,
7             others => <>);
8     end Init;
9
10    procedure Init_Some (A : out Int_Arr) is
11    begin
12      A := (1 .. 5 => 99,
13           others => 100);
14    end Init_Some;
15
16    procedure Init (A : out Int_Arr) is
17    begin
18      A := (others => 5);
19    end Init;
20
21 end Aggregates;

```

Listing 2760: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Aggregates;       use Aggregates;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42)
9      with Unreferenced;
10
11     type Test_Case_Index is
12         (Default_Rec_Chk,
13          Init_Rec_Chk,
14          Init_Some_Arr_Chk,
15          Init_Arr_Chk);
16
17     procedure Check (TC : Test_Case_Index) is
18         A : Int_Arr;
19         R : Rec;
20         DR : constant Rec := (others => <>);
21     begin
22         case TC is
23             when Default_Rec_Chk =>
24                 R := DR;
25                 Put_Line ("Record Default:");
26                 Put_Line ("W => " & Integer'Image (R.W));
27                 Put_Line ("X => " & Integer'Image (R.X));
28                 Put_Line ("Y => " & Integer'Image (R.Y));
29                 Put_Line ("Z => " & Integer'Image (R.Z));
30             when Init_Rec_Chk =>
31                 Init (R);
32                 Put_Line ("Record Init:");
33                 Put_Line ("W => " & Integer'Image (R.W));
34                 Put_Line ("X => " & Integer'Image (R.X));
35                 Put_Line ("Y => " & Integer'Image (R.Y));
36                 Put_Line ("Z => " & Integer'Image (R.Z));
37             when Init_Some_Arr_Chk =>
38                 Init_Some (A);
39                 Put_Line ("Array Init_Some:");
40                 for I in A'Range loop
41                     Put_Line (Integer'Image (I) & " "
42                               & Integer'Image (A (I)));
43                 end loop;
44             when Init_Arr_Chk =>
45                 Init (A);
46                 Put_Line ("Array Init:");
47                 for I in A'Range loop
48                     Put_Line (Integer'Image (I) & " "
49                               & Integer'Image (A (I)));
50                 end loop;
51         end case;
52     end Check;
53
54     begin
55         if Argument_Count < 1 then
56             Put_Line ("ERROR: missing arguments! Exiting...");
57             return;
58         elsif Argument_Count > 1 then
59             Put_Line ("Ignoring additional arguments...");

```

(continues on next page)

(continued from previous page)

```

60     end if;
61
62     Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

134.7.2 Versioning

Listing 2761: versioning.ads

```

1  package Versioning is
2
3      type Version is record
4          Major      : Natural;
5          Minor      : Natural;
6          Maintenance : Natural;
7      end record;
8
9      function Convert (V : Version) return String;
10
11     function Convert (V : Version) return Float;
12
13 end Versioning;

```

Listing 2762: versioning.adb

```

1  with Ada.Strings; use Ada.Strings;
2  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4  package body Versioning is
5
6      function Image_Trim (N : Natural) return String is
7          S_N : constant String := Trim (Natural'Image (N), Left);
8      begin
9          return S_N;
10     end Image_Trim;
11
12     function Convert (V : Version) return String is
13         S_Major : constant String := Image_Trim (V.Major);
14         S_Minor : constant String := Image_Trim (V.Minor);
15         S_Maint : constant String := Image_Trim (V.Maintenance);
16     begin
17         return (S_Major & "." & S_Minor & "." & S_Maint);
18     end Convert;
19
20     function Convert (V : Version) return Float is
21     begin
22         return Float (V.Major) + (Float (V.Minor) / 10.0);
23     end Convert;
24
25 end Versioning;

```

Listing 2763: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Versioning;       use Versioning;
5
6  procedure Main is

```

(continues on next page)

(continued from previous page)

```

7  type Test_Case_Index is
8      (Ver_String_Chk,
9       Ver_Float_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12      V : constant Version := (1, 3, 23);
13  begin
14      case TC is
15          when Ver_String_Chk =>
16              Put_Line (Convert (V));
17          when Ver_Float_Chk =>
18              Put_Line (Float'Image (Convert (V)));
19      end case;
20  end Check;
21
22  begin
23      if Argument_Count < 1 then
24          Put_Line ("ERROR: missing arguments! Exiting...");
25          return;
26      elsif Argument_Count > 1 then
27          Put_Line ("Ignoring additional arguments...");
28      end if;
29
30      Check (Test_Case_Index'Value (Argument (1)));
31  end Main;

```

134.7.3 Simple todo list

Listing 2764: todo_lists.ads

```

1  package Todo_Lists is
2
3      type Todo_Item is access String;
4
5      type Todo_Items is array (Positive range <>) of Todo_Item;
6
7      type Todo_List (Max_Len : Natural) is record
8          Items : Todo_Items (1 .. Max_Len);
9          Last  : Natural := 0;
10     end record;
11
12     procedure Add (Todos : in out Todo_List;
13                  Item  : String);
14
15     procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;

```

Listing 2765: todo_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                   Item  : String) is
7      begin
8          if Todos.Last < Todos.Items'Last then
9              Todos.Last := Todos.Last + 1;

```

(continues on next page)

(continued from previous page)

```

10     Todos.Items (Todos.Last) := new String'(Item);
11 else
12     Put_Line ("ERROR: list is full!");
13 end if;
14 end Add;
15
16 procedure Display (Todos : Todo_List) is
17 begin
18     Put_Line ("TO-DO LIST");
19     for I in Todos.Items'First .. Todos.Last loop
20         Put_Line (Todos.Items (I).all);
21     end loop;
22 end Display;
23
24 end Todo_Lists;

```

Listing 2766: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List (10);
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;

```

134.7.4 Price list

Listing 2767: price_lists.ads

```

1 package Price_Lists is
2
3     type Price_Type is delta 0.01 digits 12;
4
5     type Price_List_Array is array (Positive range <>) of Price_Type;
6
7     type Price_List (Max : Positive) is record
8         List : Price_List_Array (1 .. Max);
9         Last : Natural := 0;
10    end record;
11
12    type Price_Result (Ok : Boolean) is record
13        case Ok is
14            when False =>
15                null;
16            when True =>
17                Price : Price_Type;
18        end case;
19    end record;
20
21    procedure Reset (Prices : in out Price_List);
22
23    procedure Add (Prices : in out Price_List;
24                  Item   : Price_Type);
25
26    function Get (Prices : Price_List;
27                 Idx     : Positive) return Price_Result;
28
29    procedure Display (Prices : Price_List);
30
31 end Price_Lists;

```

Listing 2768: price_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Price_Lists is
4
5     procedure Reset (Prices : in out Price_List) is
6     begin
7         Prices.Last := 0;
8     end Reset;
9
10    procedure Add (Prices : in out Price_List;
11                  Item   : Price_Type) is
12    begin
13        if Prices.Last < Prices.List'Last then
14            Prices.Last := Prices.Last + 1;
15            Prices.List (Prices.Last) := Item;
16        else
17            Put_Line ("ERROR: list is full!");
18        end if;
19    end Add;
20
21    function Get (Prices : Price_List;
22                 Idx     : Positive) return Price_Result is
23    begin
24        if (Idx >= Prices.List'First and then

```

(continues on next page)

(continued from previous page)

```

25         Idx <= Prices.Last)           then
26         return Price_Result'(Ok      => True,
27                               Price => Prices.List (Idx));
28     else
29         return Price_Result'(Ok      => False);
30     end if;
31 end Get;
32
33 procedure Display (Prices : Price_List) is
34 begin
35     Put_Line ("PRICE LIST");
36     for I in Prices.List'First .. Prices.Last loop
37         Put_Line (Price_Type'Image (Prices.List (I)));
38     end loop;
39 end Display;
40
41 end Price_Lists;

```

Listing 2769: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Price_Lists;      use Price_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Price_Type_Chk,
9           Price_List_Chk,
10          Price_List_Get_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13          L : Price_List (10);
14
15          procedure Local_Init_List is
16          begin
17              Reset (L);
18              Add (L, 1.45);
19              Add (L, 2.37);
20              Add (L, 3.21);
21              Add (L, 4.14);
22              Add (L, 5.22);
23              Add (L, 6.69);
24              Add (L, 7.77);
25              Add (L, 8.14);
26              Add (L, 9.99);
27              Add (L, 10.01);
28          end Local_Init_List;
29
30          procedure Get_Display (Idx : Positive) is
31              R : constant Price_Result := Get (L, Idx);
32          begin
33              Put_Line ("Attempt Get # " & Positive'Image (Idx));
34              if R.Ok then
35                  Put_Line ("Element # " & Positive'Image (Idx)
36                          & " => " & Price_Type'Image (R.Price));
37              else
38                  declare
39                  begin
40                      Put_Line ("Element # " & Positive'Image (Idx)
41                              & " => " & Price_Type'Image (R.Price));

```

(continues on next page)

(continued from previous page)

```

42         exception
43         when others =>
44             Put_Line ("Element not available (as expected)");
45         end;
46     end if;
47
48     end Get_Display;
49
50     begin
51         case TC is
52             when Price_Type_Chk =>
53                 Put_Line ("The delta    value of Price_Type is "
54                     & Price_Type'Image (Price_Type'Delta) & ";");
55                 Put_Line ("The minimum  value of Price_Type is "
56                     & Price_Type'Image (Price_Type'First) & ";");
57                 Put_Line ("The maximum  value of Price_Type is "
58                     & Price_Type'Image (Price_Type'Last) & ";");
59             when Price_List_Chk =>
60                 Local_Init_List;
61                 Display (L);
62             when Price_List_Get_Chk =>
63                 Local_Init_List;
64                 Get_Display (5);
65                 Get_Display (40);
66         end case;
67     end Check;
68
69     begin
70         if Argument_Count < 1 then
71             Put_Line ("ERROR: missing arguments! Exiting...");
72             return;
73         elsif Argument_Count > 1 then
74             Put_Line ("Ignoring additional arguments...");
75         end if;
76
77         Check (Test_Case_Index'Value (Argument (1)));
78     end Main;

```

134.8 Privacy

134.8.1 Directions

Listing 2770: directions.ads

```

1  package Directions is
2
3      type Angle_Mod is mod 360;
4
5      type Direction is
6          (North,
7           Northwest,
8           West,
9           Southwest,
10          South,
11          Southeast,
12          East);
13
14      function To_Direction (N : Angle_Mod) return Direction;
15

```

(continues on next page)

(continued from previous page)

```

16  type Ext_Angle is private;
17
18  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
19
20  procedure Display (N : Ext_Angle);
21
22  private
23
24  type Ext_Angle is record
25      Angle_Elem      : Angle_Mod;
26      Direction_Elem : Direction;
27  end record;
28
29  end Directions;

```

Listing 2771: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0      => return East;
18              when 1 .. 89 => return Northwest;
19              when 90     => return North;
20              when 91 .. 179 => return Northwest;
21              when 180    => return West;
22              when 181 .. 269 => return Southwest;
23              when 270    => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem => N,
31                 Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34  end Directions;

```

Listing 2772: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
4      type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;
5
6      All_Directions : constant Ext_Angle_Array (1 .. 6)

```

(continues on next page)

(continued from previous page)

```

7      := (To_Ext_Angle (0),
8          To_Ext_Angle (45),
9          To_Ext_Angle (90),
10         To_Ext_Angle (91),
11         To_Ext_Angle (180),
12         To_Ext_Angle (270));
13
14 begin
15     for I in All_Directions'Range loop
16         Display (All_Directions (I));
17     end loop;
18
19 end Test_Directions;

```

Listing 2773: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Direction_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Direction_Chk =>
14                 Test_Directions;
15         end case;
16     end Check;
17
18 begin
19     if Argument_Count < 1 then
20         Put_Line ("ERROR: missing arguments! Exiting...");
21         return;
22     elsif Argument_Count > 1 then
23         Put_Line ("Ignoring additional arguments...");
24     end if;
25
26     Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

134.8.2 Limited Strings

Listing 2774: limited_strings.ads

```

1  package Limited_Strings is
2
3      type Lim_String is limited private;
4
5      function Init (S : String) return Lim_String;
6
7      function Init (Max : Positive) return Lim_String;
8
9      procedure Put_Line (LS : Lim_String);
10
11     procedure Copy (From : Lim_String;

```

(continues on next page)

(continued from previous page)

```

12         To : in out Lim_String);
13
14     function "=" (Ref, Dut : Lim_String) return Boolean;
15
16 private
17
18     type Lim_String is access String;
19
20 end Limited_Strings;

```

Listing 2775: limited_strings.adb

```

1  with Ada.Text_IO;
2
3  package body Limited_Strings
4  is
5
6      function Init (S : String) return Lim_String is
7          LS : constant Lim_String := new String'(S);
8      begin
9          return Ls;
10     end Init;
11
12     function Init (Max : Positive) return Lim_String is
13         LS : constant Lim_String := new String (1 .. Max);
14     begin
15         LS.all := (others => '_');
16         return LS;
17     end Init;
18
19     procedure Put_Line (LS : Lim_String) is
20     begin
21         Ada.Text_IO.Put_Line (LS.all);
22     end Put_Line;
23
24     function Get_Min_Last (A, B : Lim_String) return Positive is
25     begin
26         return Positive'Min (A'Last, B'Last);
27     end Get_Min_Last;
28
29     procedure Copy (From : Lim_String;
30                   To : in out Lim_String) is
31         Min_Last : constant Positive := Get_Min_Last (From, To);
32     begin
33         To (To'First .. Min_Last) := From (To'First .. Min_Last);
34         To (Min_Last + 1 .. To'Last) := (others => '_');
35     end;
36
37     function "=" (Ref, Dut : Lim_String) return Boolean is
38         Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39     begin
40         for I in Dut'First .. Min_Last loop
41             if Dut (I) /= Ref (I) then
42                 return False;
43             end if;
44         end loop;
45
46         return True;
47     end;
48
49 end Limited_Strings;

```

Listing 2776: check_lim_string.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Limited_Strings; use Limited_Strings;
4
5  procedure Check_Lim_String is
6      S : constant String := "-----";
7      S1 : constant Lim_String := Init ("Hello World");
8      S2 : constant Lim_String := Init (30);
9      S3 : Lim_String := Init (5);
10     S4 : Lim_String := Init (S & S & S);
11 begin
12     Put ("S1 => ");
13     Put_Line (S1);
14     Put ("S2 => ");
15     Put_Line (S2);
16
17     if S1 = S2 then
18         Put_Line ("S1 is equal to S2.");
19     else
20         Put_Line ("S1 isn't equal to S2.");
21     end if;
22
23     Copy (From => S1, To => S3);
24     Put ("S3 => ");
25     Put_Line (S3);
26
27     if S1 = S3 then
28         Put_Line ("S1 is equal to S3.");
29     else
30         Put_Line ("S1 isn't equal to S3.");
31     end if;
32
33     Copy (From => S1, To => S4);
34     Put ("S4 => ");
35     Put_Line (S4);
36
37     if S1 = S4 then
38         Put_Line ("S1 is equal to S4.");
39     else
40         Put_Line ("S1 isn't equal to S4.");
41     end if;
42 end Check_Lim_String;

```

Listing 2777: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Check_Lim_String;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Lim_String_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13         when Lim_String_Chk =>
14             Check_Lim_String;

```

(continues on next page)

(continued from previous page)

```

15     end case;
16 end Check;
17
18 begin
19     if Argument_Count < 1 then
20         Put_Line ("ERROR: missing arguments! Exiting...");
21         return;
22     elsif Argument_Count > 1 then
23         Put_Line ("Ignoring additional arguments...");
24     end if;
25
26     Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

134.9 Generics

134.9.1 Display Array

Listing 2778: display_array.ads

```

1 generic
2     type T_Range is range <>;
3     type T_Element is private;
4     type T_Array is array (T_Range range <>) of T_Element;
5     with function Image (E : T_Element) return String;
6     procedure Display_Array (Header : String;
7                             A      : T_Array);

```

Listing 2779: display_array.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Array (Header : String;
4                         A      : T_Array) is
5 begin
6     Put_Line (Header);
7     for I in A'Range loop
8         Put_Line (T_Range'Image (I) & ": " & Image (A (I)));
9     end loop;
10 end Display_Array;

```

Listing 2780: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Array;
5
6 procedure Main is
7     type Test_Case_Index is (Int_Array_Chk,
8                             Point_Array_Chk);
9
10    procedure Test_Int_Array is
11        type Int_Array is array (Positive range <>) of Integer;
12
13        procedure Display_Int_Array is new
14            Display_Array (T_Range => Positive,
15                          T_Element => Integer,

```

(continues on next page)

(continued from previous page)

```

16         T_Array    => Int_Array,
17         Image       => Integer'Image);
18
19     A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20 begin
21     Display_Int_Array ("Integers", A);
22 end Test_Int_Array;
23
24 procedure Test_Point_Array is
25     type Point is record
26         X : Float;
27         Y : Float;
28     end record;
29
30     type Point_Array is array (Natural range <>) of Point;
31
32     function Image (P : Point) return String is
33     begin
34         return "(" & Float'Image (P.X)
35             & ", " & Float'Image (P.Y) & ")";
36     end Image;
37
38     procedure Display_Point_Array is new
39         Display_Array (T_Range    => Natural,
40                       T_Element   => Point,
41                       T_Array     => Point_Array,
42                       Image       => Image);
43
44     A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                           (5.0, 2.0), (-0.5, 2.0));
46 begin
47     Display_Point_Array ("Points", A);
48 end Test_Point_Array;
49
50 procedure Check (TC : Test_Case_Index) is
51 begin
52     case TC is
53         when Int_Array_Chk =>
54             Test_Int_Array;
55         when Point_Array_Chk =>
56             Test_Point_Array;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

134.9.2 Average of Array of Float

Listing 2781: average.ads

```

1 generic
2   type T_Range is range <>;
3   type T_Element is digits <>;
4   type T_Array is array (T_Range range <>) of T_Element;
5   function Average (A : T_Array) return T_Element;

```

Listing 2782: average.adb

```

1 function Average (A : T_Array) return T_Element is
2   Acc : Float := 0.0;
3 begin
4   for I in A'Range loop
5     Acc := Acc + Float (A (I));
6   end loop;
7
8   return T_Element (Acc / Float (A'Length));
9 end Average;

```

Listing 2783: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Average;
5
6 procedure Main is
7   type Test_Case_Index is (Float_Array_Chk,
8                             Digits_7_Float_Array_Chk);
9
10  procedure Test_Float_Array is
11    type Float_Array is array (Positive range <>) of Float;
12
13    function Average_Float is new
14      Average (T_Range => Positive,
15               T_Element => Float,
16               T_Array => Float_Array);
17
18    A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19  begin
20    Put_Line ("Average: " & Float'Image (Average_Float (A)));
21  end Test_Float_Array;
22
23  procedure Test_Digits_7_Float_Array is
24    type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26    type Float_Array is
27      array (Integer range <>) of Custom_Float;
28
29    function Average_Float is new
30      Average (T_Range => Integer,
31               T_Element => Custom_Float,
32               T_Array => Float_Array);
33
34    A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
35  begin
36    Put_Line ("Average: "
37              & Custom_Float'Image (Average_Float (A)));
38  end Test_Digits_7_Float_Array;
39

```

(continues on next page)

(continued from previous page)

```

40  procedure Check (TC : Test_Case_Index) is
41  begin
42      case TC is
43          when Float_Array_Chk =>
44              Test_Float_Array;
45          when Digits_7_Float_Array_Chk =>
46              Test_Digits_7_Float_Array;
47      end case;
48  end Check;
49
50  begin
51      if Argument_Count < 1 then
52          Put_Line ("ERROR: missing arguments! Exiting...");
53          return;
54      elsif Argument_Count > 1 then
55          Put_Line ("Ignoring additional arguments...");
56      end if;
57
58      Check (Test_Case_Index'Value (Argument (1)));
59  end Main;

```

134.9.3 Average of Array of Any Type

Listing 2784: average.ads

```

1  generic
2      type T_Range is range <>;
3      type T_Element is private;
4      type T_Array is array (T_Range range <>) of T_Element;
5      with function To_Float (E : T_Element) return Float is <>;
6  function Average (A : T_Array) return Float;

```

Listing 2785: average.adb

```

1  function Average (A : T_Array) return Float is
2      Acc : Float := 0.0;
3  begin
4      for I in A'Range loop
5          Acc := Acc + To_Float (A (I));
6      end loop;
7
8      return Acc / Float (A'Length);
9  end Average;

```

Listing 2786: test_item.ads

```

1  procedure Test_Item;

```

Listing 2787: test_item.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Average;
4
5  procedure Test_Item is
6      package F_IO is new Ada.Text_IO.Float_IO (Float);
7
8      type Amount is delta 0.01 digits 12;
9

```

(continues on next page)

(continued from previous page)

```

10  type Item is record
11      Quantity : Natural;
12      Price    : Amount;
13  end record;
14
15  type Item_Array is
16      array (Positive range <>) of Item;
17
18  function Get_Total (I : Item) return Float is
19      (Float (I.Quantity) * Float (I.Price));
20
21  function Get_Price (I : Item) return Float is
22      (Float (I.Price));
23
24  function Average_Total is new
25      Average (T_Range => Positive,
26              T_Element => Item,
27              T_Array   => Item_Array,
28              To_Float  => Get_Total);
29
30  function Average_Price is new
31      Average (T_Range => Positive,
32              T_Element => Item,
33              T_Array   => Item_Array,
34              To_Float  => Get_Price);
35
36  A : constant Item_Array (1 .. 4)
37      := ((Quantity => 5,   Price => 10.00),
38         (Quantity => 80,  Price => 2.50),
39         (Quantity => 40,  Price => 5.00),
40         (Quantity => 20,  Price => 12.50));
41
42  begin
43      Put ("Average per item & quantity: ");
44      F_IO.Put (Average_Total (A), 3, 2, 0);
45      New_Line;
46
47      Put ("Average price: ");
48      F_IO.Put (Average_Price (A), 3, 2, 0);
49      New_Line;
50  end Test_Item;

```

Listing 2788: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Item;
5
6  procedure Main is
7      type Test_Case_Index is (Item_Array_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         begin
11             case TC is
12                 when Item_Array_Chk =>
13                     Test_Item;
14             end case;
15         end Check;
16
17  begin

```

(continues on next page)

(continued from previous page)

```

18   if Argument_Count < 1 then
19       Put_Line ("ERROR: missing arguments! Exiting...");
20       return;
21   elsif Argument_Count > 1 then
22       Put_Line ("Ignoring additional arguments...");
23   end if;
24
25   Check (Test_Case_Index'Value (Argument (1)));
26 end Main;

```

134.9.4 Generic list

Listing 2789: gen_list.ads

```

1  generic
2      type Item is private;
3      type Items is array (Positive range <>) of Item;
4      Name      : String;
5      List_Array : in out Items;
6      Last      : in out Natural;
7      with procedure Put (I : Item) is <>;
8  package Gen_List is
9
10     procedure Init;
11
12     procedure Add (I      : Item;
13                  Status : out Boolean);
14
15     procedure Display;
16
17 end Gen_List;

```

Listing 2790: gen_list.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Gen_List is
4
5      procedure Init is
6      begin
7          Last := List_Array'First - 1;
8      end Init;
9
10     procedure Add (I      : Item;
11                  Status : out Boolean) is
12     begin
13         Status := Last < List_Array'Last;
14
15         if Status then
16             Last := Last + 1;
17             List_Array (Last) := I;
18         end if;
19     end Add;
20
21     procedure Display is
22     begin
23         Put_Line (Name);
24         for I in List_Array'First .. Last loop
25             Put (List_Array (I));

```

(continues on next page)

(continued from previous page)

```

26     New_Line;
27     end loop;
28 end Display;
29
30 end Gen_List;

```

Listing 2791: test_int.ads

```

1 procedure Test_Int;

```

Listing 2792: test_int.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_List;
4
5 procedure Test_Int is
6
7     procedure Put (I : Integer) is
8     begin
9         Ada.Text_IO.Put (Integer'Image (I));
10    end Put;
11
12    type Integer_Array is array (Positive range <>) of Integer;
13
14    A : Integer_Array (1 .. 3);
15    L : Natural;
16
17    package Int_List is new
18        Gen_List (Item      => Integer,
19                  Items     => Integer_Array,
20                  Name      => "List of integers",
21                  List_Array => A,
22                  Last      => L);
23
24    Success : Boolean;
25
26    procedure Display_Add_Success (Success : Boolean) is
27    begin
28        if Success then
29            Put_Line ("Added item successfully!");
30        else
31            Put_Line ("Couldn't add item!");
32        end if;
33
34    end Display_Add_Success;
35
36 begin
37     Int_List.Init;
38
39     Int_List.Add (2, Success);
40     Display_Add_Success (Success);
41
42     Int_List.Add (5, Success);
43     Display_Add_Success (Success);
44
45     Int_List.Add (7, Success);
46     Display_Add_Success (Success);
47
48     Int_List.Add (8, Success);

```

(continues on next page)

(continued from previous page)

```

49     Display_Add_Success (Success);
50
51     Int_List.Display;
52 end Test_Int;

```

Listing 2793: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Int;
5
6  procedure Main is
7      type Test_Case_Index is (Int_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         begin
11             case TC is
12                 when Int_Chk =>
13                     Test_Int;
14             end case;
15         end Check;
16
17     begin
18         if Argument_Count < 1 then
19             Put_Line ("ERROR: missing arguments! Exiting...");
20             return;
21         elsif Argument_Count > 1 then
22             Put_Line ("Ignoring additional arguments...");
23         end if;
24
25         Check (Test_Case_Index'Value (Argument (1)));
26     end Main;

```

134.10 Exceptions

134.10.1 Uninitialized Value

Listing 2794: options.ads

```

1  package Options is
2
3      type Option is (Uninitialized,
4                      Option_1,
5                      Option_2,
6                      Option_3);
7
8      Uninitialized_Value : exception;
9
10     function Image (O : Option) return String;
11
12 end Options;

```

Listing 2795: options.adb

```

1  package body Options is
2
3      function Image (O : Option) return String is

```

(continues on next page)

(continued from previous page)

```

4   begin
5       case 0 is
6           when Uninitialized =>
7               raise Uninitialized_Value with "Uninitialized value detected!";
8           when others =>
9               return Option'Image (0);
10          end case;
11      end Image;
12
13  end Options;

```

Listing 2796: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Exceptions;   use Ada.Exceptions;
4
5  with Options;          use Options;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Options_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12
13         procedure Check (O : Option) is
14             begin
15                 Put_Line (Image (O));
16             exception
17                 when E : Uninitialized_Value =>
18                     Put_Line (Exception_Message (E));
19             end Check;
20
21     begin
22         case TC is
23             when Options_Chk =>
24                 for O in Option loop
25                     Check (O);
26                 end loop;
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;

```

134.10.2 Numerical Exception

Listing 2797: tests.ads

```

1  package Tests is
2

```

(continues on next page)

(continued from previous page)

```

3  type Test_ID is (Test_1, Test_2);
4
5  Custom_Exception : exception;
6
7  procedure Num_Exception_Test (ID : Test_ID);
8
9  end Tests;

```

Listing 2798: tests.adb

```

1  package body Tests is
2
3      pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5      procedure Num_Exception_Test (ID : Test_ID) is
6          A, B, C : Integer;
7      begin
8          case ID is
9              when Test_1 =>
10                 A := Integer'Last;
11                 B := Integer'Last;
12                 C := A + B;
13              when Test_2 =>
14                 raise Custom_Exception with "Custom_Exception raised!";
15          end case;
16      end Num_Exception_Test;
17
18      pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20  end Tests;

```

Listing 2799: check_exception.adb

```

1  with Tests;          use Tests;
2
3  with Ada.Text_IO;    use Ada.Text_IO;
4  with Ada.Exceptions; use Ada.Exceptions;
5
6  procedure Check_Exception (ID : Test_ID) is
7      begin
8          Num_Exception_Test (ID);
9      exception
10         when Constraint_Error =>
11             Put_Line ("Constraint_Error detected!");
12         when E : others =>
13             Put_Line (Exception_Message (E));
14  end Check_Exception;

```

Listing 2800: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Exceptions;   use Ada.Exceptions;
4
5  with Tests;            use Tests;
6  with Check_Exception;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Exception_1_Chk,

```

(continues on next page)

(continued from previous page)

```

11     Exception_2_Chk);
12
13 procedure Check (TC : Test_Case_Index) is
14
15     procedure Check_Handle_Exception (ID : Test_ID) is
16     begin
17         Check_Exception (ID);
18     exception
19         when Constraint_Error =>
20             Put_Line ("Constraint_Error"
21                 & " (raised by Check_Exception) detected!");
22         when E : others =>
23             Put_Line (Exception_Name (E)
24                 & " (raised by Check_Exception) detected!");
25     end Check_Handle_Exception;
26
27 begin
28     case TC is
29     when Exception_1_Chk =>
30         Check_Handle_Exception (Test_1);
31     when Exception_2_Chk =>
32         Check_Handle_Exception (Test_2);
33     end case;
34 end Check;
35
36 begin
37     if Argument_Count < 1 then
38         Put_Line ("ERROR: missing arguments! Exiting...");
39         return;
40     elsif Argument_Count > 1 then
41         Put_Line ("Ignoring additional arguments...");
42     end if;
43
44     Check (Test_Case_Index'Value (Argument (1)));
45 end Main;

```

134.10.3 Re-raising Exceptions

Listing 2801: tests.ads

```

1 package Tests is
2
3     type Test_ID is (Test_1, Test_2);
4
5     Custom_Exception, Another_Exception : exception;
6
7     procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 2802: tests.adb

```

1 package body Tests is
2
3     pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5     procedure Num_Exception_Test (ID : Test_ID) is
6         A, B, C : Integer;
7     begin

```

(continues on next page)

(continued from previous page)

```

8      case ID is
9          when Test_1 =>
10             A := Integer'Last;
11             B := Integer'Last;
12             C := A + B;
13             when Test_2 =>
14                 raise Custom_Exception with "Custom_Exception raised!";
15             end case;
16     end Num_Exception_Test;
17
18     pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;

```

Listing 2803: check_exception.ads

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID);

```

Listing 2804: check_exception.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Check_Exception (ID : Test_ID) is
5 begin
6     Num_Exception_Test (ID);
7 exception
8     when Constraint_Error =>
9         Put_Line ("Constraint_Error detected!");
10        raise;
11     when E : others =>
12         Put_Line (Exception_Message (E));
13         raise Another_Exception;
14 end Check_Exception;

```

Listing 2805: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with Ada.Exceptions; use Ada.Exceptions;
4
5 with Tests; use Tests;
6 with Check_Exception;
7
8 procedure Main is
9     type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13     procedure Check (TC : Test_Case_Index) is
14
15         procedure Check_Handle_Exception (ID : Test_ID) is
16         begin
17             Check_Exception (ID);
18         exception
19             when Constraint_Error =>
20                 Put_Line ("Constraint_Error"
21                     & " (raised by Check_Exception) detected!");

```

(continues on next page)

(continued from previous page)

```

22     when E : others =>
23         Put_Line (Exception_Name (E)
24             & " (raised by Check_Exception) detected!");
25     end Check_Handle_Exception;
26
27     begin
28         case TC is
29         when Exception_1_Chk =>
30             Check_Handle_Exception (Test_1);
31         when Exception_2_Chk =>
32             Check_Handle_Exception (Test_2);
33         end case;
34     end Check;
35
36     begin
37         if Argument_Count < 1 then
38             Put_Line ("ERROR: missing arguments! Exiting...");
39             return;
40         elsif Argument_Count > 1 then
41             Put_Line ("Ignoring additional arguments...");
42         end if;
43
44         Check (Test_Case_Index'Value (Argument (1)));
45     end Main;

```

134.11 Tasking

134.11.1 Display Service

Listing 2806: display_services.ads

```

1  package Display_Services is
2
3      task type Display_Service is
4          entry Display (S : String);
5          entry Display (I : Integer);
6      end Display_Service;
7
8  end Display_Services;

```

Listing 2807: display_services.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Display_Services is
4
5      task body Display_Service is
6      begin
7          loop
8              select
9                  accept Display (S : String) do
10                     Put_Line (S);
11                 end Display;
12             or
13                 accept Display (I : Integer) do
14                     Put_Line (Integer'Image (I));
15                 end Display;
16             or

```

(continues on next page)

(continued from previous page)

```

17         terminate;
18     end select;
19 end loop;
20 end Display_Service;
21
22 end Display_Services;

```

Listing 2808: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Display_Services; use Display_Services;
5
6  procedure Main is
7      type Test_Case_Index is (Display_Service_Chk);
8
9      procedure Check (TC : Test_Case_Index) is
10         Display : Display_Service;
11     begin
12         case TC is
13             when Display_Service_Chk =>
14                 Display.Display ("Hello");
15                 delay 0.5;
16                 Display.Display ("Hello again");
17                 delay 0.5;
18                 Display.Display (55);
19                 delay 0.5;
20         end case;
21     end Check;
22
23     begin
24         if Argument_Count < 1 then
25             Put_Line ("ERROR: missing arguments! Exiting...");
26             return;
27         elsif Argument_Count > 1 then
28             Put_Line ("Ignoring additional arguments...");
29         end if;
30
31         Check (Test_Case_Index'Value (Argument (1)));
32     end Main;

```

134.11.2 Event Manager

Listing 2809: event_managers.ads

```

1  with Ada.Real_Time; use Ada.Real_Time;
2
3  package Event_Managers is
4
5      task type Event_Manager is
6          entry Start (ID : Natural);
7          entry Event (T : Time);
8      end Event_Manager;
9
10 end Event_Managers;

```

Listing 2810: event_managers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Event_Managers is
4
5      task body Event_Manager is
6          Event_ID    : Natural := 0;
7          Event_Delay : Time;
8      begin
9          accept Start (ID : Natural) do
10             Event_ID := ID;
11          end Start;
12
13          accept Event (T : Time) do
14             Event_Delay := T;
15          end Event;
16
17          delay until Event_Delay;
18
19          Put_Line ("Event #" & Natural'Image (Event_ID));
20      end Event_Manager;
21
22 end Event_Managers;

```

Listing 2811: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Event_Managers;   use Event_Managers;
5  with Ada.Real_Time;    use Ada.Real_Time;
6
7  procedure Main is
8      type Test_Case_Index is (Event_Manager_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         Ev_Mng : array (1 .. 5) of Event_Manager;
12     begin
13         case TC is
14             when Event_Manager_Chk =>
15                 for I in Ev_Mng'Range loop
16                     Ev_Mng (I).Start (I);
17                 end loop;
18                 Ev_Mng (1).Event (Clock + Seconds (5));
19                 Ev_Mng (2).Event (Clock + Seconds (3));
20                 Ev_Mng (3).Event (Clock + Seconds (1));
21                 Ev_Mng (4).Event (Clock + Seconds (2));
22                 Ev_Mng (5).Event (Clock + Seconds (4));
23             end case;
24     end Check;
25
26     begin
27         if Argument_Count < 1 then
28             Put_Line ("ERROR: missing arguments! Exiting...");
29             return;
30         elsif Argument_Count > 1 then
31             Put_Line ("Ignoring additional arguments...");
32         end if;
33
34         Check (Test_Case_Index'Value (Argument (1)));

```

(continues on next page)

```
35 end Main;
```

134.11.3 Generic Protected Queue

Listing 2812: gen_queues.ads

```

1 generic
2   type Queue_Index is mod <>;
3   type T is private;
4   package Gen_Queues is
5
6     type Queue_Array is array (Queue_Index) of T;
7
8     protected type Queue is
9       function Empty return Boolean;
10      function Full return Boolean;
11      entry Push (V : T);
12      entry Pop (V : out T);
13    private
14      N : Natural := 0;
15      Idx : Queue_Index := Queue_Array'First;
16      A : Queue_Array;
17    end Queue;
18
19 end Gen_Queues;
```

Listing 2813: gen_queues.adb

```

1 package body Gen_Queues is
2
3   protected body Queue is
4
5     function Empty return Boolean is
6       (N = 0);
7
8     function Full return Boolean is
9       (N = A'Length);
10
11    entry Push (V : T) when not Full is
12    begin
13      A (Idx) := V;
14
15      Idx := Idx + 1;
16      N := N + 1;
17    end Push;
18
19    entry Pop (V : out T) when not Empty is
20    begin
21      N := N - 1;
22
23      V := A (Idx - Queue_Index (N) - 1);
24    end Pop;
25
26  end Queue;
27
28 end Gen_Queues;
```

Listing 2814: queue_tests.ads

```

1 package Queue_Tests is
2
3     procedure Simple_Test;
4
5     procedure Concurrent_Test;
6
7 end Queue_Tests;
```

Listing 2815: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7     Max : constant := 10;
8     type Queue_Mod is mod Max;
9
10    procedure Simple_Test is
11        package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13        Q_F : Queues_Float.Queue;
14        V : Float;
15    begin
16        V := 10.0;
17        while not Q_F.Full loop
18            Q_F.Push (V);
19            V := V + 1.5;
20        end loop;
21
22        while not Q_F.Empty loop
23            Q_F.Pop (V);
24            Put_Line ("Value from queue: " & Float'Image (V));
25        end loop;
26    end Simple_Test;
27
28    procedure Concurrent_Test is
29        package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
30
31        Q_I : Queues_Integer.Queue;
32
33        task T_Producer;
34        task T_Consumer;
35
36        task body T_Producer is
37            V : Integer := 100;
38        begin
39            for I in 1 .. 2 * Max loop
40                Q_I.Push (V);
41                V := V + 1;
42            end loop;
43        end T_Producer;
44
45        task body T_Consumer is
46            V : Integer;
47        begin
48            delay 1.5;
```

(continues on next page)

(continued from previous page)

```

50     while not Q_I.Empty loop
51         Q_I.Pop (V);
52         Put_Line ("Value from queue: " & Integer'Image (V));
53         delay 0.2;
54     end loop;
55 end T_Consumer;
56 begin
57     null;
58 end Concurrent_Test;
59
60 end Queue_Tests;

```

Listing 2816: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Queue_Tests;       use Queue_Tests;
5
6  procedure Main is
7      type Test_Case_Index is (Simple_Queue_Chk,
8                               Concurrent_Queue_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11     begin
12         case TC is
13             when Simple_Queue_Chk =>
14                 Simple_Test;
15             when Concurrent_Queue_Chk =>
16                 Concurrent_Test;
17         end case;
18     end Check;
19
20     begin
21         if Argument_Count < 1 then
22             Put_Line ("ERROR: missing arguments! Exiting...");
23             return;
24         elsif Argument_Count > 1 then
25             Put_Line ("Ignoring additional arguments...");
26         end if;
27
28         Check (Test_Case_Index'Value (Argument (1)));
29     end Main;

```

134.12 Design by contracts

134.12.1 Price Range

Listing 2817: prices.ads

```

1  package Prices is
2
3      type Amount is delta 10.0 ** (-2) digits 12;
4
5      -- subtype Price is Amount range 0.0 .. Amount'Last;
6
7      subtype Price is Amount
8          with Static_Predicate => Price >= 0.0;

```

(continues on next page)

(continued from previous page)

```

9
10 end Prices;

```

Listing 2818: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Prices;           use Prices;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Price_Range_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
13
14         procedure Check_Range (A : Amount) is
15             P : constant Price := A;
16         begin
17             Put_Line ("Price: " & Price'Image (P));
18         end Check_Range;
19
20     begin
21         case TC is
22             when Price_Range_Chk =>
23                 Check_Range (-2.0);
24             end case;
25     exception
26         when Constraint_Error =>
27             Put_Line ("Constraint_Error detected (NOT as expected).");
28         when Assert_Failure =>
29             Put_Line ("Assert_Failure detected (as expected).");
30     end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting...");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments...");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

134.12.2 Pythagorean Theorem: Predicate

Listing 2819: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;

```

(continues on next page)

(continued from previous page)

```

9      -- Catheti / legs
10     end record
11     with Dynamic_Predicate => H * H = C1 * C1 + C2 * C2;
12
13     function Init (H, C1, C2 : Length) return Right_Triangle is
14         ((H, C1, C2));
15
16 end Triangles;

```

Listing 2820: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 2821: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         "(" & Length'Image (T.H)
5         & ", " & Length'Image (T.C1)
6         & ", " & Length'Image (T.C2)
7         & ")";
8
9 end Triangles.IO;

```

Listing 2822: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles; use Triangles;
6 with Triangles.IO; use Triangles.IO;
7
8 procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");

```

(continues on next page)

(continued from previous page)

```

30     end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk  => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;

```

134.12.3 Pythagorean Theorem: Precondition

Listing 2823: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record
6          H      : Length := 0;
7          -- Hypotenuse
8          C1, C2 : Length := 0;
9          -- Catheti / legs
10     end record;
11
12     function Init (H, C1, C2 : Length) return Right_Triangle is
13         ((H, C1, C2))
14         with Pre => H * H = C1 * C1 + C2 * C2;
15
16 end Triangles;

```

Listing 2824: triangles-io.ads

```

1  package Triangles.IO is
2
3      function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 2825: triangles-io.adb

```

1  package body Triangles.IO is
2
3      function Image (T : Right_Triangle) return String is
4          "(" & Length'Image (T.H)

```

(continues on next page)

(continued from previous page)

```

5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9  end Triangles.IO;
```

Listing 2826: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Triangles;        use Triangles;
6  with Triangles.IO;     use Triangles.IO;
7
8  procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");
30             end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk  => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;
```

134.12.4 Pythagorean Theorem: Postcondition

Listing 2827: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10  end record;
11
12  function Init (H, C1, C2 : Length) return Right_Triangle is
13    ((H, C1, C2))
14    with Post => (Init'Result.H * Init'Result.H
15                 = Init'Result.C1 * Init'Result.C1
16                 + Init'Result.C2 * Init'Result.C2);
17
18 end Triangles;
```

Listing 2828: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;
```

Listing 2829: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     (" " & Length'Image (T.H)
5     & ", " & Length'Image (T.C1)
6     & ", " & Length'Image (T.C2)
7     & " ");
8
9 end Triangles.IO;
```

Listing 2830: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10  type Test_Case_Index is
11    (Triangle_8_6_Pass_Chk,
12     Triangle_8_6_Fail_Chk,
13     Triangle_10_24_Pass_Chk,
14     Triangle_10_24_Fail_Chk,
15     Triangle_18_24_Pass_Chk,
16     Triangle_18_24_Fail_Chk);
```

(continues on next page)

(continued from previous page)

```

17
18 procedure Check (TC : Test_Case_Index) is
19
20     procedure Check_Triangle (H, C1, C2 : Length) is
21         T : Right_Triangle;
22     begin
23         T := Init (H, C1, C2);
24         Put_Line (Image (T));
25     exception
26         when Constraint_Error =>
27             Put_Line ("Constraint_Error detected (NOT as expected).");
28         when Assert_Failure =>
29             Put_Line ("Assert_Failure detected (as expected).");
30     end Check_Triangle;
31
32 begin
33     case TC is
34         when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35         when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36         when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37         when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38         when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39         when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40     end case;
41 end Check;
42
43 begin
44     if Argument_Count < 1 then
45         Put_Line ("ERROR: missing arguments! Exiting...");
46         return;
47     elsif Argument_Count > 1 then
48         Put_Line ("Ignoring additional arguments...");
49     end if;
50
51     Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

134.12.5 Pythagorean Theorem: Type Invariant

Listing 2831: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is private
6         with Type_Invariant => Check (Right_Triangle);
7
8     function Check (T : Right_Triangle) return Boolean;
9
10    function Init (H, C1, C2 : Length) return Right_Triangle;
11
12 private
13
14    type Right_Triangle is record
15        H : Length := 0;
16        -- Hypotenuse
17        C1, C2 : Length := 0;
18        -- Catheti / legs
19    end record;

```

(continues on next page)

(continued from previous page)

```

20
21  function Init (H, C1, C2 : Length) return Right_Triangle is
22      ((H, C1, C2));
23
24  function Check (T : Right_Triangle) return Boolean is
25      (T.H * T.H = T.C1 * T.C1 + T.C2 * T.C2);
26
27  end Triangles;

```

Listing 2832: triangles-io.ads

```

1  package Triangles.IO is
2
3      function Image (T : Right_Triangle) return String;
4
5  end Triangles.IO;

```

Listing 2833: triangles-io.adb

```

1  package body Triangles.IO is
2
3      function Image (T : Right_Triangle) return String is
4          "(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")";
8
9  end Triangles.IO;

```

Listing 2834: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Triangles; use Triangles;
6  with Triangles.IO; use Triangles.IO;
7
8  procedure Main is
9
10     type Test_Case_Index is
11         (Triangle_8_6_Pass_Chk,
12          Triangle_8_6_Fail_Chk,
13          Triangle_10_24_Pass_Chk,
14          Triangle_10_24_Fail_Chk,
15          Triangle_18_24_Pass_Chk,
16          Triangle_18_24_Fail_Chk);
17
18     procedure Check (TC : Test_Case_Index) is
19
20         procedure Check_Triangle (H, C1, C2 : Length) is
21             T : Right_Triangle;
22             begin
23                 T := Init (H, C1, C2);
24                 Put_Line (Image (T));
25             exception
26                 when Constraint_Error =>
27                     Put_Line ("Constraint_Error detected (NOT as expected).");
28                 when Assert_Failure =>
29                     Put_Line ("Assert_Failure detected (as expected).");

```

(continues on next page)

(continued from previous page)

```

30     end Check_Triangle;
31
32     begin
33         case TC is
34             when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35             when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36             when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37             when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38             when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39             when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40         end case;
41     end Check;
42
43     begin
44         if Argument_Count < 1 then
45             Put_Line ("ERROR: missing arguments! Exiting...");
46             return;
47         elsif Argument_Count > 1 then
48             Put_Line ("Ignoring additional arguments...");
49         end if;
50
51         Check (Test_Case_Index'Value (Argument (1)));
52     end Main;

```

134.12.6 Primary Colors

Listing 2835: color_types.ads

```

1  package Color_Types is
2
3      type HTML_Color is
4          (Salmon,
5           Firebrick,
6           Red,
7           Darkred,
8           Lime,
9           Forestgreen,
10          Green,
11          Darkgreen,
12          Blue,
13          Mediumblue,
14          Darkblue);
15
16      subtype Int_Color is Integer range 0 .. 255;
17
18      function Image (I : Int_Color) return String;
19
20      type RGB is record
21          Red   : Int_Color;
22          Green : Int_Color;
23          Blue  : Int_Color;
24      end record;
25
26      function To_RGB (C : HTML_Color) return RGB;
27
28      function Image (C : RGB) return String;
29
30      type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32      To_RGB_Lookup_Table : constant HTML_Color_RGB_Array

```

(continues on next page)

(continued from previous page)

```

33     := (Salmon      => (16#FA#, 16#80#, 16#72#),
34         Firebrick  => (16#B2#, 16#22#, 16#22#),
35         Red         => (16#FF#, 16#00#, 16#00#),
36         Darkred     => (16#8B#, 16#00#, 16#00#),
37         Lime        => (16#00#, 16#FF#, 16#00#),
38         Forestgreen => (16#22#, 16#8B#, 16#22#),
39         Green       => (16#00#, 16#80#, 16#00#),
40         Darkgreen   => (16#00#, 16#64#, 16#00#),
41         Blue        => (16#00#, 16#00#, 16#FF#),
42         Mediumblue  => (16#00#, 16#00#, 16#CD#),
43         Darkblue    => (16#00#, 16#00#, 16#8B#));
44
45     subtype HTML_RGB_Color is HTML_Color
46         with Static_Predicate => HTML_RGB_Color in Red | Green | Blue;
47
48     function To_Int_Color (C : HTML_Color;
49                           S : HTML_RGB_Color) return Int_Color;
50     -- Convert to hexadecimal value for the selected RGB component S
51
52 end Color_Types;

```

Listing 2836: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_RGB (C : HTML_Color) return RGB is
6      begin
7          return To_RGB_Lookup_Table (C);
8      end To_RGB;
9
10     function To_Int_Color (C : HTML_Color;
11                           S : HTML_RGB_Color) return Int_Color is
12     C_RGB : constant RGB := To_RGB (C);
13     begin
14         case S is
15             when Red    => return C_RGB.Red;
16             when Green => return C_RGB.Green;
17             when Blue  => return C_RGB.Blue;
18         end case;
19     end To_Int_Color;
20
21     function Image (I : Int_Color) return String is
22     subtype Str_Range is Integer range 1 .. 10;
23     S : String (Str_Range);
24     begin
25         Ada.Integer_Text_IO.Put (To    => S,
26                                Item  => I,
27                                Base  => 16);
28
29         return S;
30     end Image;
31
32     function Image (C : RGB) return String is
33     begin
34         return ("(Red => " & Image (C.Red)
35                & ", Green => " & Image (C.Green)
36                & ", Blue => " & Image (C.Blue)
37                & ")");
38     end Image;

```

(continues on next page)

(continued from previous page)

39 `end Color_Types;`

Listing 2837: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_Red_Chk,
9           HTML_Color_Green_Chk,
10          HTML_Color_Blue_Chk);
11
12      procedure Check (TC : Test_Case_Index) is
13
14          procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15              begin
16                  Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17                  for I in HTML_Color'Range loop
18                      Put_Line (HTML_Color'Image (I) & " => "
19                              & Image (To_Int_Color (I, S)) & ".");
20                  end loop;
21              end Check_HTML_Colors;
22
23          begin
24              case TC is
25                  when HTML_Color_Red_Chk =>
26                      Check_HTML_Colors (Red);
27                  when HTML_Color_Green_Chk =>
28                      Check_HTML_Colors (Green);
29                  when HTML_Color_Blue_Chk =>
30                      Check_HTML_Colors (Blue);
31              end case;
32          end Check;
33
34      begin
35          if Argument_Count < 1 then
36              Put_Line ("ERROR: missing arguments! Exiting...");
37              return;
38          elsif Argument_Count > 1 then
39              Put_Line ("Ignoring additional arguments...");
40          end if;
41
42          Check (Test_Case_Index'Value (Argument (1)));
43      end Main;

```

134.13 Object-oriented programming

134.13.1 Simple type extension

Listing 2838: type_extensions.ads

```

1  package Type_Extensions is
2
3      type T_Float is tagged record
4          F : Float;

```

(continues on next page)

(continued from previous page)

```

5  end record;
6
7  function Init (F : Float) return T_Float;
8
9  function Init (I : Integer) return T_Float;
10
11 function Image (T : T_Float) return String;
12
13 type T_Mixed is new T_Float with record
14     I : Integer;
15 end record;
16
17 function Init (F : Float) return T_Mixed;
18
19 function Init (I : Integer) return T_Mixed;
20
21 function Image (T : T_Mixed) return String;
22
23 end Type_Extensions;

```

Listing 2839: type_extensions.adb

```

1  package body Type_Extensions is
2
3      function Init (F : Float) return T_Float is
4      begin
5          return ((F => F));
6      end Init;
7
8      function Init (I : Integer) return T_Float is
9      begin
10         return ((F => Float (I)));
11     end Init;
12
13     function Init (F : Float) return T_Mixed is
14     begin
15         return ((F => F,
16                 I => Integer (F)));
17     end Init;
18
19     function Init (I : Integer) return T_Mixed is
20     begin
21         return ((F => Float (I),
22                 I => I));
23     end Init;
24
25     function Image (T : T_Float) return String is
26     begin
27         return "{ F => " & Float'Image (T.F) & " }";
28     end Image;
29
30     function Image (T : T_Mixed) return String is
31     begin
32         return "{ F => " & Float'Image (T.F)
33             & ", I => " & Integer'Image (T.I) & " }";
34     end Image;
35
36 end Type_Extensions;

```

Listing 2840: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Type_Extensions; use Type_Extensions;
5
6  procedure Main is
7
8      type Test_Case_Index is
9          (Type_Extension_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         F1, F2 : T_Float;
13         M1, M2 : T_Mixed;
14     begin
15         case TC is
16             when Type_Extension_Chk =>
17                 F1 := Init (2.0);
18                 F2 := Init (3);
19                 M1 := Init (4.0);
20                 M2 := Init (5);
21
22                 if M2 in T_Float'Class then
23                     Put_Line ("T_Mixed is in T_Float'Class as expected");
24                 end if;
25
26                 Put_Line ("F1: " & Image (F1));
27                 Put_Line ("F2: " & Image (F2));
28                 Put_Line ("M1: " & Image (M1));
29                 Put_Line ("M2: " & Image (M2));
30             end case;
31         end Check;
32
33     begin
34         if Argument_Count < 1 then
35             Put_Line ("ERROR: missing arguments! Exiting...");
36             return;
37         elsif Argument_Count > 1 then
38             Put_Line ("Ignoring additional arguments...");
39         end if;
40
41         Check (Test_Case_Index'Value (Argument (1)));
42     end Main;

```

134.13.2 Online Store

Listing 2841: online_store.ads

```

1  with Ada.Calendar; use Ada.Calendar;
2
3  package Online_Store is
4
5      type Amount is delta 10.0**(-2) digits 10;
6
7      subtype Percentage is Amount range 0.0 .. 1.0;
8
9      type Member is tagged record
10         Start : Year_Number;
11     end record;

```

(continues on next page)

(continued from previous page)

```

12
13  type Member_Access is access Member'Class;
14
15  function Get_Status (M : Member) return String;
16
17  function Get_Price (M : Member;
18                    P : Amount) return Amount;
19
20  type Full_Member is new Member with record
21    Discount : Percentage;
22  end record;
23
24  function Get_Status (M : Full_Member) return String;
25
26  function Get_Price (M : Full_Member;
27                    P : Amount) return Amount;
28
29  end Online_Store;

```

Listing 2842: online_store.adb

```

1  package body Online_Store is
2
3    function Get_Status (M : Member) return String is
4      ("Associate Member");
5
6    function Get_Status (M : Full_Member) return String is
7      ("Full Member");
8
9    function Get_Price (M : Member;
10                     P : Amount) return Amount is (P);
11
12    function Get_Price (M : Full_Member;
13                     P : Amount) return Amount is
14      (P * (1.0 - M.Discount));
15
16  end Online_Store;

```

Listing 2843: online_store-tests.ads

```

1  package Online_Store.Tests is
2
3    procedure Simple_Test;
4
5  end Online_Store.Tests;

```

Listing 2844: online_store-tests.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Online_Store.Tests is
4
5    procedure Simple_Test is
6
7      type Member_Due_Amount is record
8        Member      : Member_Access;
9        Due_Amount : Amount;
10     end record;
11
12     function Get_Price (MA : Member_Due_Amount) return Amount is

```

(continues on next page)

(continued from previous page)

```

13   begin
14       return MA.Member.Get_Price (MA.Due_Amount);
15   end Get_Price;
16
17   type Member_Due_Amounts is array (Positive range <>) of Member_Due_Amount;
18
19   DB : constant Member_Due_Amounts (1 .. 4)
20       := ((Member      => new Member'(Start => 2010),
21           Due_Amount => 250.0),
22          (Member      => new Full_Member'(Start  => 1998,
23                                           Discount => 0.1),
24           Due_Amount => 160.0),
25          (Member      => new Full_Member'(Start  => 1987,
26                                           Discount => 0.2),
27           Due_Amount => 400.0),
28          (Member      => new Member'(Start => 2013),
29           Due_Amount => 110.0));
30   begin
31       for I in DB'Range loop
32           Put_Line ("Member #" & Positive'Image (I));
33           Put_Line ("Status: " & DB (I).Member.Get_Status);
34           Put_Line ("Since: " & Year_Number'Image (DB (I).Member.Start));
35           Put_Line ("Due Amount: " & Amount'Image (Get_Price (DB (I))));
36           Put_Line ("-----");
37       end loop;
38   end Simple_Test;
39
40 end Online_Store.Tests;

```

Listing 2845: main.adb

```

1  with Ada.Command_Line;   use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Online_Store;        use Online_Store;
5  with Online_Store.Tests;  use Online_Store.Tests;
6
7  procedure Main is
8
9      type Test_Case_Index is
10         (Type_Chk,
11          Unit_Test_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14
15         function Result_Image (Result : Boolean) return String is
16             (if Result then "OK" else "not OK");
17
18         begin
19             case TC is
20                 when Type_Chk =>
21                     declare
22                         AM : constant Member      := (Start  => 2002);
23                         FM : constant Full_Member := (Start  => 1990,
24                                                         Discount => 0.2);
25                     begin
26                         Put_Line ("Testing Status of Associate Member Type => "
27                                & Result_Image (AM.Get_Status = "Associate Member"));
28                         Put_Line ("Testing Status of Full Member Type => "
29                                & Result_Image (FM.Get_Status = "Full Member"));
30                         Put_Line ("Testing Discount of Associate Member Type => "

```

(continues on next page)

(continued from previous page)

```

31         & Result_Image (AM.Get_Price (100.0) = 100.0));
32     Put_Line ("Testing Discount of Full Member Type => "
33         & Result_Image (FM.Get_Price (100.0) = 80.0));
34     end;
35     when Unit_Test_Chk =>
36         Simple_Test;
37     end case;
38 end Check;
39
40 begin
41     if Argument_Count < 1 then
42         Put_Line ("ERROR: missing arguments! Exiting...");
43         return;
44     elsif Argument_Count > 1 then
45         Put_Line ("Ignoring additional arguments...");
46     end if;
47
48     Check (Test_Case_Index'Value (Argument (1)));
49 end Main;

```

134.14 Standard library: Containers

134.14.1 Simple todo list

Listing 2846: todo_lists.ads

```

1  with Ada.Containers.Vectors;
2
3  package Todo_Lists is
4
5      type Todo_Item is access String;
6
7      package Todo_List_Pkg is new Ada.Containers.Vectors
8          (Index_Type => Natural,
9           Element_Type => Todo_Item);
10
11     subtype Todo_List is Todo_List_Pkg.Vector;
12
13     procedure Add (Todos : in out Todo_List;
14                 Item : String);
15
16     procedure Display (Todos : Todo_List);
17
18 end Todo_Lists;

```

Listing 2847: todo_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                  Item : String) is
7      begin
8          Todos.Append (new String'(Item));
9      end Add;
10
11     procedure Display (Todos : Todo_List) is

```

(continues on next page)

(continued from previous page)

```
12   begin
13       Put_Line ("TO-DO LIST");
14       for T of Todos loop
15           Put_Line (T.all);
16       end loop;
17   end Display;
18
19 end Todo_Lists;
```

Listing 2848: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List;
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreadsheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27             end case;
28     end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting...");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments...");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;
```

134.14.2 List of unique integers

Listing 2849: ops.ads

```
1  with Ada.Containers.Ordered_Sets;
2
3  package Ops is
4
```

(continues on next page)

(continued from previous page)

```

5  type Int_Array is array (Positive range <>) of Integer;
6
7  package Integer_Sets is new Ada.Containers.Ordered_Sets
8    (Element_Type => Integer);
9
10 subtype Int_Set is Integer_Sets.Set;
11
12 function Get_Unique (A : Int_Array) return Int_Set;
13
14 function Get_Unique (A : Int_Array) return Int_Array;
15
16 end Ops;

```

Listing 2850: ops.adb

```

1  package body Ops is
2
3    function Get_Unique (A : Int_Array) return Int_Set is
4      S : Int_Set;
5    begin
6      for E of A loop
7        S.Include (E);
8      end loop;
9
10     return S;
11   end Get_Unique;
12
13   function Get_Unique (A : Int_Array) return Int_Array is
14     S : constant Int_Set := Get_Unique (A);
15     AR : Int_Array (1 .. Positive (S.Length));
16     I : Positive := 1;
17   begin
18     for E of S loop
19       AR (I) := E;
20       I := I + 1;
21     end loop;
22
23     return AR;
24   end Get_Unique;
25
26 end Ops;

```

Listing 2851: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Ops;                  use Ops;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Get_Unique_Set_Chk,
9       Get_Unique_Array_Chk);
10
11    procedure Check (TC : Test_Case_Index;
12                   A : Int_Array) is
13
14      procedure Display_Unique_Set (A : Int_Array) is
15        S : constant Int_Set := Get_Unique (A);
16      begin

```

(continues on next page)

(continued from previous page)

```

17     for E of S loop
18         Put_Line (Integer'Image (E));
19     end loop;
20 end Display_Unique_Set;
21
22 procedure Display_Unique_Array (A : Int_Array) is
23     AU : constant Int_Array := Get_Unique (A);
24 begin
25     for E of AU loop
26         Put_Line (Integer'Image (E));
27     end loop;
28 end Display_Unique_Array;
29
30 begin
31     case TC is
32         when Get_Unique_Set_Chk => Display_Unique_Set (A);
33         when Get_Unique_Array_Chk => Display_Unique_Array (A);
34     end case;
35 end Check;
36
37 begin
38     if Argument_Count < 3 then
39         Put_Line ("ERROR: missing arguments! Exiting...");
40         return;
41     else
42         declare
43             A : Int_Array (1 .. Argument_Count - 1);
44         begin
45             for I in A'Range loop
46                 A (I) := Integer'Value (Argument (1 + I));
47             end loop;
48             Check (Test_Case_Index'Value (Argument (1)), A);
49         end;
50     end if;
51 end Main;

```

134.15 Standard library: Dates & Times

134.15.1 Holocene calendar

Listing 2852: to_holocene_year.adb

```

1 with Ada.Calendar; use Ada.Calendar;
2
3 function To_Holocene_Year (T : Time) return Integer is
4 begin
5     return Year (T) + 10_000;
6 end To_Holocene_Year;

```

Listing 2853: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Calendar;          use Ada.Calendar;
4
5 with To_Holocene_Year;
6
7 procedure Main is

```

(continues on next page)

(continued from previous page)

```

8  type Test_Case_Index is
9      (Holocene_Chk);
10
11  procedure Display_Holocene_Year (Y : Year_Number) is
12      HY : Integer;
13  begin
14      HY := To_Holocene_Year (Time_Of (Y, 1, 1));
15      Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
16      Put_Line ("Year (Holocene): " & Integer'Image (HY));
17  end Display_Holocene_Year;
18
19  procedure Check (TC : Test_Case_Index) is
20  begin
21      case TC is
22          when Holocene_Chk =>
23              Display_Holocene_Year (2012);
24              Display_Holocene_Year (2020);
25      end case;
26  end Check;
27
28  begin
29      if Argument_Count < 1 then
30          Put_Line ("ERROR: missing arguments! Exiting...");
31          return;
32      elsif Argument_Count > 1 then
33          Put_Line ("Ignoring additional arguments...");
34      end if;
35
36      Check (Test_Case_Index'Value (Argument (1)));
37  end Main;

```

134.15.2 List of events

Listing 2854: events.ads

```

1  with Ada.Containers.Vectors;
2
3  package Events is
4
5      type Event_Item is access String;
6
7      package Event_Item_Containers is new
8          Ada.Containers.Vectors
9              (Index_Type => Positive,
10               Element_Type => Event_Item);
11
12      subtype Event_Items is Event_Item_Containers.Vector;
13
14  end Events;

```

Listing 2855: events-lists.ads

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Containers.Ordered_Maps;
3
4  package Events.Lists is
5
6      type Event_List is tagged private;
7

```

(continues on next page)

(continued from previous page)

```

8  procedure Add (Events      : in out Event_List;
9                  Event_Time :      Time;
10                 Event       :      String);
11
12  procedure Display (Events : Event_List);
13
14  private
15
16  package Event_Time_Item_Containers is new
17    Ada.Containers.Ordered_Maps
18    (Key_Type      => Time,
19     Element_Type  => Event_Items,
20     "="          => Event_Item_Containers."=");
21
22  type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24  end Events.Lists;

```

Listing 2856: events-lists.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6  procedure Add (Events      : in out Event_List;
7                  Event_Time : Time;
8                  Event       : String) is
9    use Event_Item_Containers;
10   E : constant Event_Item := new String'(Event);
11  begin
12    if not Events.Contains (Event_Time) then
13      Events.Include (Event_Time, Empty_Vector);
14    end if;
15    Events (Event_Time).Append (E);
16  end Add;
17
18  function Date_Image (T : Time) return String is
19    Date_Img : constant String := Image (T);
20  begin
21    return Date_Img (1 .. 10);
22  end;
23
24  procedure Display (Events : Event_List) is
25    use Event_Time_Item_Containers;
26    T : Time;
27  begin
28    Put_Line ("EVENTS LIST");
29    for C in Events.Iterate loop
30      T := Key (C);
31      Put_Line ("- " & Date_Image (T));
32      for I of Events (C) loop
33        Put_Line ("  - " & I.all);
34      end loop;
35    end loop;
36  end Display;
37
38  end Events.Lists;

```

Listing 2857: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6  with Events.Lists;           use Events.Lists;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Event_List_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
13         EL : Event_List;
14     begin
15         case TC is
16             when Event_List_Chk =>
17                 EL.Add (Time_Of (2018, 2, 16),
18                     "Final check");
19                 EL.Add (Time_Of (2018, 2, 16),
20                     "Release");
21                 EL.Add (Time_Of (2018, 12, 3),
22                     "Brother's birthday");
23                 EL.Add (Time_Of (2018, 1, 1),
24                     "New Year's Day");
25                 EL.Display;
26         end case;
27     end Check;
28
29 begin
30     if Argument_Count < 1 then
31         Put_Line ("ERROR: missing arguments! Exiting...");
32         return;
33     elsif Argument_Count > 1 then
34         Put_Line ("Ignoring additional arguments...");
35     end if;
36
37     Check (Test_Case_Index'Value (Argument (1)));
38 end Main;

```

134.16 Standard library: Strings

134.16.1 Concatenation

Listing 2858: str_concat.ads

```

1  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3  package Str_Concat is
4
5      type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7      function Concat (USA           : Unbounded_Strings;
8                      Trim_Str      : Boolean;
9                      Add_Whitespace : Boolean) return Unbounded_String;
10
11     function Concat (USA           : Unbounded_Strings;
12                     Trim_Str      : Boolean;

```

(continues on next page)

(continued from previous page)

```

13         Add_Whitespace : Boolean) return String;
14
15 end Str_Concat;

```

Listing 2859: str_concat.adb

```

1  with Ada.Strings; use Ada.Strings;
2
3  package body Str_Concat is
4
5      function Concat (USA           : Unbounded_Strings;
6                      Trim_Str      : Boolean;
7                      Add_Whitespace : Boolean) return Unbounded_String is
8
9          function Retrieve (USA           : Unbounded_Strings;
10                           Trim_Str      : Boolean;
11                           Index         : Positive) return Unbounded_String is
12              US_Internal : Unbounded_String := USA (Index);
13          begin
14              if Trim_Str then
15                  US_Internal := Trim (US_Internal, Both);
16              end if;
17              return US_Internal;
18          end Retrieve;
19
20          US : Unbounded_String := To_Unbounded_String ("");
21      begin
22          for I in USA'First .. USA'Last - 1 loop
23              US := US & Retrieve (USA, Trim_Str, I);
24              if Add_Whitespace then
25                  US := US & " ";
26              end if;
27          end loop;
28          US := US & Retrieve (USA, Trim_Str, USA'Last);
29
30          return US;
31      end Concat;
32
33      function Concat (USA           : Unbounded_Strings;
34                      Trim_Str      : Boolean;
35                      Add_Whitespace : Boolean) return String is
36      begin
37          return To_String (Concat (USA, Trim_Str, Add_Whitespace));
38      end Concat;
39
40 end Str_Concat;

```

Listing 2860: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
4
5  with Str_Concat;            use Str_Concat;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Unbounded_Concat_No_Trim_No_WS_Chk,
10           Unbounded_Concat_Trim_No_WS_Chk,
11           String_Concat_Trim_WS_Chk,

```

(continues on next page)

(continued from previous page)

```

12     Concat_Single_Element);
13
14 procedure Check (TC : Test_Case_Index) is
15 begin
16     case TC is
17     when Unbounded_Concat_No_Trim_No_WS_Chk =>
18         declare
19             S : constant Unbounded_Strings := (
20                 To_Unbounded_String ("Hello"),
21                 To_Unbounded_String (" World"),
22                 To_Unbounded_String ("!"));
23         begin
24             Put_Line (To_String (Concat (S, False, False)));
25         end;
26     when Unbounded_Concat_Trim_No_WS_Chk =>
27         declare
28             S : constant Unbounded_Strings := (
29                 To_Unbounded_String (" This "),
30                 To_Unbounded_String (" _is_ "),
31                 To_Unbounded_String (" a "),
32                 To_Unbounded_String (" _check "));
33         begin
34             Put_Line (To_String (Concat (S, True, False)));
35         end;
36     when String_Concat_Trim_WS_Chk =>
37         declare
38             S : constant Unbounded_Strings := (
39                 To_Unbounded_String (" This "),
40                 To_Unbounded_String (" is a "),
41                 To_Unbounded_String (" test. "));
42         begin
43             Put_Line (Concat (S, True, True));
44         end;
45     when Concat_Single_Element =>
46         declare
47             S : constant Unbounded_Strings := (
48                 1 => To_Unbounded_String (" Hi "));
49         begin
50             Put_Line (Concat (S, True, True));
51         end;
52     end case;
53 end Check;
54
55 begin
56     if Argument_Count < 1 then
57         Put_Line ("ERROR: missing arguments! Exiting...");
58         return;
59     elsif Argument_Count > 1 then
60         Put_Line ("Ignoring additional arguments...");
61     end if;
62
63     Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

134.16.2 List of events

Listing 2861: events.ads

```

1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2 with Ada.Containers.Vectors;

```

(continues on next page)

(continued from previous page)

```

3
4 package Events is
5
6     subtype Event_Item is Unbounded_String;
7
8     package Event_Item_Containers is new
9         Ada.Containers.Vectors
10         (Index_Type => Positive,
11          Element_Type => Event_Item);
12
13     subtype Event_Items is Event_Item_Containers.Vector;
14
15 end Events;
```

Listing 2862: events-lists.ads

```

1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Containers.Ordered_Maps;
3
4 package Events.Lists is
5
6     type Event_List is tagged private;
7
8     procedure Add (Events      : in out Event_List;
9                  Event_Time   : Time;
10                  Event        : String);
11
12     procedure Display (Events : Event_List);
13
14 private
15
16     package Event_Time_Item_Containers is new
17         Ada.Containers.Ordered_Maps
18         (Key_Type      => Time,
19          Element_Type   => Event_Items,
20          "="            => Event_Item_Containers "=");
21
22     type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;
```

Listing 2863: events-lists.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4 package body Events.Lists is
5
6     procedure Add (Events      : in out Event_List;
7                  Event_Time   : Time;
8                  Event        : String) is
9         use Event_Item_Containers;
10        E : constant Event_Item := To_Unbounded_String (Event);
11    begin
12        if not Events.Contains (Event_Time) then
13            Events.Include (Event_Time, Empty_Vector);
14        end if;
15        Events (Event_Time).Append (E);
16    end Add;
17
```

(continues on next page)

(continued from previous page)

```

18  function Date_Image (T : Time) return String is
19      Date_Img : constant String := Image (T);
20  begin
21      return Date_Img (1 .. 10);
22  end;
23
24  procedure Display (Events : Event_List) is
25      use Event_Time_Item_Containers;
26      T : Time;
27  begin
28      Put_Line ("EVENTS LIST");
29      for C in Events.Iterate loop
30          T := Key (C);
31          Put_Line ("- " & Date_Image (T));
32          for I of Events (C) loop
33              Put_Line ("    - " & To_String (I));
34          end loop;
35      end loop;
36  end Display;
37
38  end Events.Lists;

```

Listing 2864: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;          use Events.Lists;
9
10 procedure Main is
11     type Test_Case_Index is
12         (Unbounded_String_Chk,
13          Event_List_Chk);
14
15     procedure Check (TC : Test_Case_Index) is
16         EL : Event_List;
17     begin
18         case TC is
19             when Unbounded_String_Chk =>
20                 declare
21                     S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22                 begin
23                     Put_Line (To_String (S));
24                 end;
25             when Event_List_Chk =>
26                 EL.Add (Time_Of (2018, 2, 16),
27                     "Final check");
28                 EL.Add (Time_Of (2018, 2, 16),
29                     "Release");
30                 EL.Add (Time_Of (2018, 12, 3),
31                     "Brother's birthday");
32                 EL.Add (Time_Of (2018, 1, 1),
33                     "New Year's Day");
34                 EL.Display;
35             end case;
36     end Check;
37

```

(continues on next page)

(continued from previous page)

```
38 begin
39   if Argument_Count < 1 then
40     Put_Line ("ERROR: missing arguments! Exiting...");
41     return;
42   elsif Argument_Count > 1 then
43     Put_Line ("Ignoring additional arguments...");
44   end if;
45
46   Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

134.17 Standard library: Numerics

134.17.1 Decibel Factor

Listing 2865: decibels.ads

```
1 package Decibels is
2
3   subtype Decibel is Float;
4   subtype Factor is Float;
5
6   function To_Decibel (F : Factor) return Decibel;
7
8   function To_Factor (D : Decibel) return Factor;
9
10 end Decibels;
```

Listing 2866: decibels.adb

```
1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Decibels is
4
5   function To_Decibel (F : Factor) return Decibel is
6   begin
7     return 20.0 * Log (F, 10.0);
8   end To_Decibel;
9
10  function To_Factor (D : Decibel) return Factor is
11  begin
12    return 10.0 ** (D / 20.0);
13  end To_Factor;
14
15 end Decibels;
```

Listing 2867: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Decibels; use Decibels;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Db_Chk,
9      Factor_Chk);
10
```

(continues on next page)

(continued from previous page)

```

11  procedure Check (TC : Test_Case_Index; V : Float) is
12
13      package F_IO is new Ada.Text_IO.Float_IO (Factor);
14      package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16      procedure Put_Decibel_Cnvt (D : Decibel) is
17          F : constant Factor := To_Factor (D);
18      begin
19          D_IO.Put (D, 0, 2, 0);
20          Put (" dB => Factor of ");
21          F_IO.Put (F, 0, 2, 0);
22          New_Line;
23      end;
24
25      procedure Put_Factor_Cnvt (F : Factor) is
26          D : constant Decibel := To_Decibel (F);
27      begin
28          Put ("Factor of ");
29          F_IO.Put (F, 0, 2, 0);
30          Put (" => ");
31          D_IO.Put (D, 0, 2, 0);
32          Put_Line (" dB");
33      end;
34  begin
35      case TC is
36          when Db_Chk =>
37              Put_Decibel_Cnvt (Decibel (V));
38          when Factor_Chk =>
39              Put_Factor_Cnvt (Factor (V));
40      end case;
41  end Check;
42
43  begin
44      if Argument_Count < 2 then
45          Put_Line ("ERROR: missing arguments! Exiting...");
46          return;
47      elsif Argument_Count > 2 then
48          Put_Line ("Ignoring additional arguments...");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52  end Main;

```

134.17.2 Root-Mean-Square

Listing 2868: signals.ads

```

1  package Signals is
2
3      subtype Sig_Value is Float;
4
5      type Signal is array (Natural range <>) of Sig_Value;
6
7      function Rms (S : Signal) return Sig_Value;
8
9  end Signals;

```

Listing 2869: signals.adb

```
1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Signals is
4
5     function Rms (S : Signal) return Sig_Value is
6         Acc : Float := 0.0;
7     begin
8         for V of S loop
9             Acc := Acc + V * V;
10        end loop;
11
12        return Sqrt (Acc / Float (S'Length));
13    end;
14
15 end Signals;
```

Listing 2870: signals-std.ads

```
1 package Signals.Std is
2
3     Sample_Rate : Float := 8000.0;
4
5     function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7     function Generate_Square (N : Positive) return Signal;
8
9     function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;
```

Listing 2871: signals-std.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4 package body Signals.Std is
5
6     function Generate_Sine (N : Positive; Freq : Float) return Signal is
7         S : Signal (0 .. N - 1);
8     begin
9         for I in S'First .. S'Last loop
10             S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11        end loop;
12
13        return S;
14    end;
15
16    function Generate_Square (N : Positive) return Signal is
17        S : constant Signal (0 .. N - 1) := (others => 1.0);
18    begin
19        return S;
20    end;
21
22    function Generate_Triangular (N : Positive) return Signal is
23        S : Signal (0 .. N - 1);
24        S_Half : constant Natural := S'Last / 2;
25    begin
26        for I in S'First .. S_Half loop
27            S (I) := 1.0 * (Float (I) / Float (S_Half));
```

(continues on next page)

(continued from previous page)

```

28     end loop;
29     for I in S_Half .. S'Last loop
30         S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31     end loop;
32
33     return S;
34 end;
35
36 end Signals.Std;

```

Listing 2872: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Signals;               use Signals;
5  with Signals.Std;           use Signals.Std;
6
7  procedure Main is
8      type Test_Case_Index is
9          (Sine_Signal_Chk,
10           Square_Signal_Chk,
11           Triangular_Signal_Chk);
12
13      procedure Check (TC : Test_Case_Index) is
14          package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
15
16          N : constant Positive := 1024;
17          S_Si : constant Signal := Generate_Sine (N, 440.0);
18          S_Sq : constant Signal := Generate_Square (N);
19          S_Tr : constant Signal := Generate_Triangular (N + 1);
20      begin
21          case TC is
22              when Sine_Signal_Chk =>
23                  Put ("RMS of Sine Signal: ");
24                  Sig_IO.Put (Rms (S_Si), 0, 2, 0);
25                  New_Line;
26              when Square_Signal_Chk =>
27                  Put ("RMS of Square Signal: ");
28                  Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
29                  New_Line;
30              when Triangular_Signal_Chk =>
31                  Put ("RMS of Triangular Signal: ");
32                  Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
33                  New_Line;
34          end case;
35      end Check;
36
37      begin
38          if Argument_Count < 1 then
39              Put_Line ("ERROR: missing arguments! Exiting...");
40              return;
41          elsif Argument_Count > 1 then
42              Put_Line ("Ignoring additional arguments...");
43          end if;
44
45          Check (Test_Case_Index'Value (Argument (1)));
46      end Main;

```

134.17.3 Rotation

Listing 2873: rotation.ads

```
1 with Ada.Numerics.Complex_Types;
2 use Ada.Numerics.Complex_Types;
3
4 package Rotation is
5
6     type Complex_Points is array (Positive range <>) of Complex;
7
8     function Rotation (N : Positive) return Complex_Points;
9
10 end Rotation;
```

Listing 2874: rotation.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2
3 package body Rotation is
4
5     function Rotation (N : Positive) return Complex_Points is
6         C_Angle : constant Complex :=
7             Compose_From_Polar (1.0, 2.0 * Pi / Float (N));
8     begin
9         return C : Complex_Points (1 .. N + 1) do
10             C (1) := Compose_From_Cartesian (1.0, 0.0);
11
12             for I in C'First + 1 .. C'Last loop
13                 C (I) := C (I - 1) * C_Angle;
14             end loop;
15         end return;
16     end;
17
18 end Rotation;
```

Listing 2875: angles.ads

```
1 with Rotation; use Rotation;
2
3 package Angles is
4
5     subtype Angle is Float;
6
7     type Angles is array (Positive range <>) of Angle;
8
9     function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;
```

Listing 2876: angles.adb

```
1 with Ada.Numerics; use Ada.Numerics;
2 with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4 package body Angles is
5
6     function To_Angles (C : Complex_Points) return Angles is
7     begin
8         return A : Angles (C'Range) do
9             for I in A'Range loop
```

(continues on next page)

(continued from previous page)

```

10         A (I) := Argument (C (I)) / Pi * 180.0;
11     end loop;
12 end return;
13 end To_Angles;
14
15 end Angles;

```

Listing 2877: rotation-tests.ads

```

1 package Rotation.Tests is
2
3     procedure Test_Rotation (N : Positive);
4
5     procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;

```

Listing 2878: rotation-tests.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO;
3 with Ada.Numerics;         use Ada.Numerics;
4
5 with Angles;               use Angles;
6
7 package body Rotation.Tests is
8
9     package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10    package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12    --
13    -- Adapt value due to floating-point inaccuracies
14    --
15
16    function Adapt (C : Complex) return Complex is
17        function Check_Zero (F : Float) return Float is
18            (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19        begin
20            return C_Out : Complex := C do
21                C_Out.Re := Check_Zero (C_Out.Re);
22                C_Out.Im := Check_Zero (C_Out.Im);
23            end return;
24        end Adapt;
25
26    function Adapt (A : Angle) return Angle is
27        (if A <= -179.99 and A >= -180.01 then 180.0 else A);
28
29    procedure Test_Rotation (N : Positive) is
30        C : constant Complex_Points := Rotation (N);
31    begin
32        Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
33        for V of C loop
34            Put ("Point: ");
35            C_IO.Put (Adapt (V), 0, 1, 0);
36            New_Line;
37        end loop;
38    end Test_Rotation;
39
40    procedure Test_Angles (N : Positive) is
41        C : constant Complex_Points := Rotation (N);

```

(continues on next page)

(continued from previous page)

```

42     A : constant Angles.Angles := To_Angles (C);
43 begin
44     Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
45     for V of A loop
46         Put ("Angle: ");
47         F_IO.Put (Adapt (V), 0, 2, 0);
48         Put_Line (" degrees");
49     end loop;
50 end Test_Angles;
51
52 end Rotation.Tests;

```

Listing 2879: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Rotation.Tests;         use Rotation.Tests;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Rotation_Chk,
9           Angles_Chk);
10
11     procedure Check (TC : Test_Case_Index; N : Positive) is
12     begin
13         case TC is
14             when Rotation_Chk =>
15                 Test_Rotation (N);
16             when Angles_Chk =>
17                 Test_Angles (N);
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 2 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 2 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30 end Main;

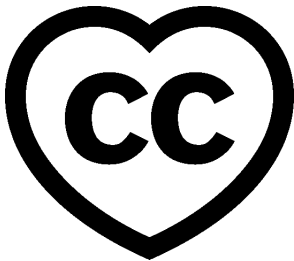
```

Part XIII

Bug Free Coding with SPARK Ada

Copyright © 2018 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](http://creativecommons.org/licenses/by-sa/4.0)⁵⁸⁹



Workshop project: Learn to write maintainable bug-free code with SPARK Ada.

This document was written by Robert Tice.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

⁵⁸⁹ <http://creativecommons.org/licenses/by-sa/4.0>

LET'S BUILD A STACK

In this lab we will build a stack data structure and use the SPARK provers to find the errors in the below implementation.

135.1 Background

So, what is a stack?

A stack is like a pile of dishes...



1. The pile starts out empty.
2. You add (push) a new plate (data) to the stack by placing it on the top of the pile.
3. To get plates (data) out, you take the one off the top of the pile (pop).
4. Our stack has a maximum height (size) of 9 dishes

Pushing items onto the stack

Here's what should happen if we pushed the string MLH onto the stack.

Step 0:
Empty

1:	
2:	
3:	
4:	
5:	

Last = 0

Step 1:
Push("M")

1:	M
2:	
3:	
4:	
5:	

Last = 1

Step 2:
Push("L")

1:	M
2:	L
3:	
4:	
5:	

Last = 2

Step 3:
Push("H")

1:	M
2:	L
3:	H
4:	
5:	

Last = 3

Step 4:
Top()

1:	M
2:	L
3:	H
4:	
5:	

Last = 3
returns:
 'H'

The list starts out empty. Each time we push a character onto the stack, Last increments by 1.

Popping items from the stack

Here's what should happen if we popped 2 characters off our stack & then clear it.

Step 0:

Start

1:	M
2:	L
3:	H
4:	
5:	

Last = 3

Step 1:

Pop()

1:	M
2:	L
3:	H
4:	
5:	

Last = 2

returns:

'H'

Step 2:

Pop()

1:	M
2:	L
3:	H
4:	
5:	

Last = 1

returns:

'L'

Step 3:

Clear()

1:	M
2:	L
3:	H
4:	
5:	

Last = 0

Note that pop and clear don't unset the Storage array's elements, they just change the value of Last.

135.2 Input Format

N inputs will be read from stdin/console as inputs, C to the stack.

135.3 Constraints

$1 \leq N \leq 1000$

C is any character. Characters d and p will be special characters corresponding to the below commands:

p => Pops a character off the stack

d => Prints the current characters in the stack

135.4 Output Format

If the stack currently has the characters "M", "L", and "H" then the program should print the stack like this:

[M, L, H]

135.5 Sample Input

M L H d p d p d p d

135.6 Sample Output

[M, L, H] [M, L] [M] []

Listing 2880: stack.ads

```
1 package Stack with SPARK_Mode => On is
2
3   procedure Push (V : Character)
4     with Pre => not Full,
5           Post => Size = Size'Old + 1;
6
7   procedure Pop (V : out Character)
8     with Pre => not Empty,
9           Post => Size = Size'Old - 1;
10
11  procedure Clear
12    with Post => Size = 0;
13
14  function Top return Character
15    with Post => Top'Result = Tab(Last);
16
17  Max_Size : constant := 9;
18  -- The stack size.
19
20  Last : Integer range 0 .. Max_Size := 0;
21  -- Indicates the top of the stack. When 0 the stack is empty.
22
23  Tab : array (1 .. Max_Size) of Character;
24  -- The stack. We push and pop pointers to Values.
25
26  function Full return Boolean is (Last = Max_Size);
27
28  function Empty return Boolean is (Last < 1);
```

(continues on next page)

(continued from previous page)

```

29
30     function Size return Integer is (Last);
31
32 end Stack;

```

Listing 2881: stack.adb

```

1  package body Stack with SPARK_Mode => On is
2
3      -----
4      -- Clear --
5      -----
6
7      procedure Clear
8      is
9      begin
10         Last := Tab'First;
11     end Clear;
12
13     -----
14     -- Push --
15     -----
16
17     procedure Push (V : Character)
18     is
19     begin
20         Tab (Last) := V;
21     end Push;
22
23     -----
24     -- Pop --
25     -----
26
27     procedure Pop (V : out Character)
28     is
29     begin
30         Last := Last - 1;
31         V := Tab (Last);
32     end Pop;
33
34     -----
35     -- Top --
36     -----
37
38     function Top return Character
39     is
40     begin
41         return Tab (1);
42     end Top;
43
44 end Stack;

```

Listing 2882: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Stack;            use Stack;
4
5  procedure Main with SPARK_Mode => Off
6  is

```

(continues on next page)

(continued from previous page)

```
7
8  -----
9  -- Debug --
10 -----
11
12 procedure Debug
13 is
14 begin
15
16     if not Stack.Empty then
17
18         Put ("[";
19         for I in Stack.Tab'First .. Stack.Size - 1 loop
20             Put (Stack.Tab (I) & ", ");
21         end loop;
22         Put_Line (Stack.Tab (Stack.Size) & "]");
23     else
24         Put_Line ("[]");
25     end if;
26
27 end Debug;
28
29 S : Character;
30
31 begin
32
33     -----
34     -- Main --
35     -----
36
37     for Arg in 1 .. Argument_Count loop
38         if Argument (Arg)'Length /= 1 then
39             Put_Line (Argument (Arg) & " is an invalid input to the stack.");
40         else
41             S := Argument (Arg)(Argument (Arg)'First);
42
43             if S = 'd' then
44                 Debug;
45             elsif S = 'p' then
46                 if not Stack.Empty then
47                     Stack.Pop (S);
48                 else
49                     Put_Line ("Nothing to Pop, Stack is empty!");
50                 end if;
51             else
52                 if not Stack.Full then
53                     Stack.Push (S);
54                 else
55                     Put_Line ("Could not push '" & S & "', Stack is full!");
56                 end if;
57             end if;
58         end if;
59
60     end loop;
61
62 end Main;
```

BIBLIOGRAPHY

- [Jorvik] A New Ravenscar-Based Profile by P. Rogers, J. Ruiz, T. Gingold and P. Bernardi, in *Reliable Software Technologies — Ada Europe 2017*, Springer-Verlag Lecture Notes in Computer Science, Number 10300.
- [SEI-C] The Software Engineering Institute. *SEI CERT C Coding Standard*.
- [MISRA2013] MISRA. 2015. *Guidelines for the Use of the C Language in Critical Systems*
- [Holzmann2006] Holzmann, G. J. 2006. *The Power of 10: Rules for Developing Safety-Critical Code*
- [ISO2000] ISO/IEC High Integrity Rapporteur Group. 2000. "ISO/IEC TR 15942:2000 Guide for the Use of the Ada Programming Language in High Integrity Systems." *ISO/IEC TR 15942:2000*, July
- [AdaRM2016] ISO/IEC. 2016. *ISO/IEC JTC 1/SC 22/WG9 Ada Reference Manual - Language and Standard Libraries-ISO/IEC 8652:2012/Cor 1:2016*
- [AdaRM2020] ISO/IEC. 2020. *ISO/IEC JTC 1/SC 22/WG9 Ada Reference Manual - Language and Standard Libraries-ISO/IEC 8652:2020*
- [AdaOOP2016] AdaCore. 2016. *High-Integrity Object-Oriented Programming in Ada, Version 1.4*⁵²⁷
- [LiskovWing1994] Liskov, B. and Wing, J. 1994. "A Behavioral Notion of Subtyping." *ACM Transactions on Programming Languages and Systems (TOPLAS)* Vol. 16, Issue 6 (November): 1811-1841.
- [DO178C] RTCA DO-178C/EUROCAE ED-12C. 2011. *Software Considerations in Airborne Systems and Equipment Certification*
- [Meyer1997] Meyer, B. 1997. "Object-Oriented Software Construction." *Prentice Hall Professional Technical Reference* (2nd Edition)
- [MITRE_CWE] MITRE. *Common Weakness Enumeration (CWE)*⁵²⁸
- [SEI-Java] The Software Engineering Institute. *SEI CERT Oracle Coding Standard for Java*
- [TR24772] ISO/IEC. 2022. *ISO/IEC TR 24772-2:20 Programming Languages - Guidance to Avoiding Vulnerabilities in Programming Languages - Part 2: Ada*

⁵²⁷ <https://www.adacore.com/uploads/techPapers/HighIntegrityAda.pdf>

⁵²⁸ <https://cwe.mitre.org/index.html>

C

Common Weakness Enumeration

CWE 248, 2052, 2054, 2056
CWE 252, 2043
CWE 362, 2021, 2024
CWE 366, 2021, 2024
CWE 367, 2021, 2024
CWE 396, 2034
CWE 397, 2050
CWE 401, 2003, 2005, 2006, 2008, 2009, 2016
CWE 415, 2003, 2006, 2008, 2009, 2014
CWE 416, 2003, 2006, 2008, 2009, 2014
CWE 457, 2032
CWE 459, 2008, 2010
CWE 478, 2028, 2031
CWE 496, 2078
CWE 547, 2047
CWE 563, 2043
CWE 567, 2025
CWE 590, 2015
CWE 667, 2025
CWE 670, 2074
CWE 674, 2044
CWE 685, 2065
CWE 754, 2040, 2074
CWE 758, 2005
CWE 770, 2011
CWE 771, 2005, 2006, 2008
CWE 789, 2011
CWE 843, 2046
CWE 1061, 2078
CWE 1074, 2061
CWE 1086, 2061
CWE 1106, 2047
CWE 1127, 2075
CWE 1325, 2005, 2006, 2008