Introduction to
GNAT Toolchain

Gustavo A. Hoffmann

LEARN.
ADACORE.COM

Introduction to GNAT Toolchain

Release 2025-12

Gustavo A. Hoffmann

Dec 27, 2025

CONTENTS:
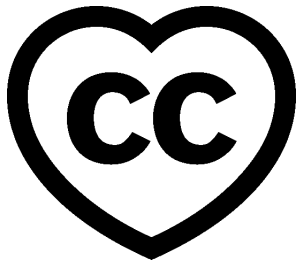
This course presents an introduction to the GNAT toolchain. The course includes first steps to get started with the toolchain and some details on the project manager (GPRbuild) and the integrated development environment (GNAT Studio).

This document was written by Gustavo A. Hoffmann, with contributions and review from Richard Kenner and Robert Duff.

> **ℹ 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.

---

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

GNAT TOOLCHAIN BASICS

This chapter presents a couple of basic commands from the GNAT toolchain.

1.1 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 the 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).

1.2 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.

1.2.1 -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 starting ignoring warnings and there'll be lots of things that generates 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:

```ada
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:

```ada
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[2] 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:

```ada
package Warnings_Example is

   procedure Process (X : in out Integer;
                      B :    out Boolean);

end Warnings_Example;
```

```ada
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;
```

```ada
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:

```ada
-- ...
begin
   Process (X, Success);

   if Success then
      Put_Line (Integer'Image (X));
   else
      Put_Line ("Couldn't process variable X.");
   end if;
end Main;
```

---

[2] 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.

1.2.2 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[3] of the GNAT User's Guide, using this switch "is equivalent to `-gnaty3aAbcefhiklmnprst`, 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.

---

[3] 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[4].

2.1 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
```

2.2 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.

2.2.1 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.

---

[4] 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).

2.2.2 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.

2.3 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.

2.3.1 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:

```ada
package Test_Pkg is

   type T is record
      X : Integer;
      Y : Integer;
   end record;

   function Init return T;
```

```ada
end Test_Pkg;
```

```ada
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:

```ada
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:

```ada
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.

2.3.2 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.

2.4 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.

2.5 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[5]. Also, you can refer to the GNAT Studio product page[6] 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.

3.1 Start-up

The first step is to start-up the GNAT Studio. The actual step depends on your platform.

3.1.1 Windows

- You may find an icon (shortcut to **GNAT Studio**) on your desktop.
- Otherwise, start **GNAT Studio** by typing `gnatstudio` on the command prompt.

3.1.2 Linux

- Start **GNAT Studio** by typing `gnatstudio` on a shell.

3.2 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.

---

[5] https://docs.adacore.com/live/wave/gps/html/gps_tutorial/index.html
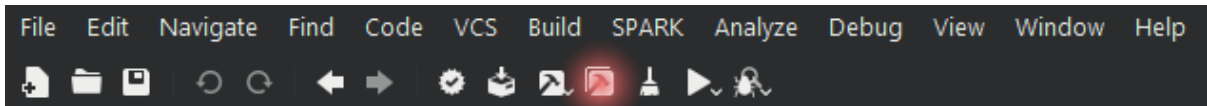[6] https://www.adacore.com/gnatpro/toolsuite/gps
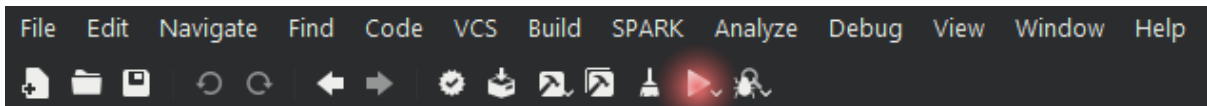
You should now have a working project file.

3.3 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`.

3.4 Debugging

3.4.1 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 7)).

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:
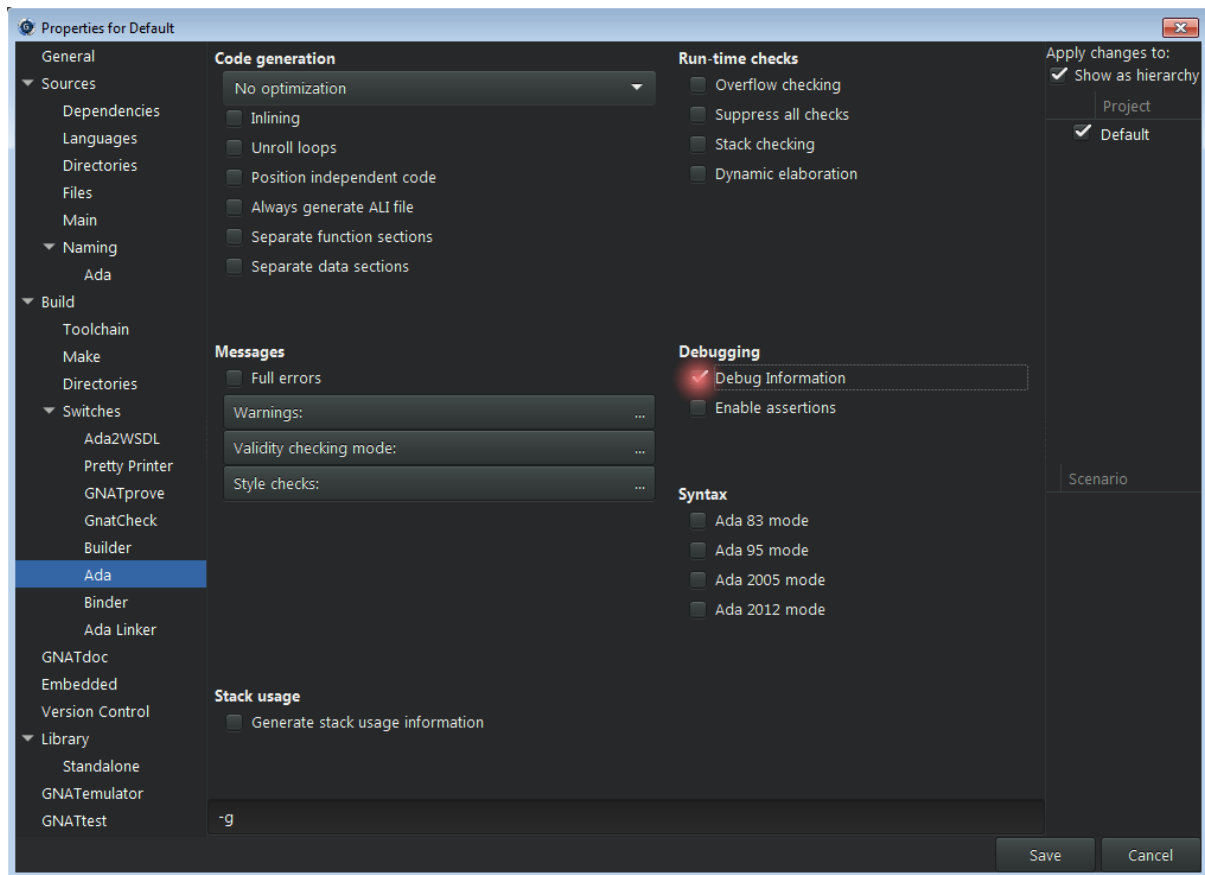
Click on `Build → Switches → Ada` on this window, and make sure that the `Debug Information` option is selected.

3.4.2 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:

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
   Put_Line ("Hello World!");
   Put_Line ("Hello again!");
end Main;
```

3.4.3 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`.

3.5 Formal verification

In order to see how SPARK can detect issues, let's creating a simple application that accumulates values in a variable A:

```ada
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:

```ada
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);
```

<div align="right">(continues on next page)</div>

```
   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[7].

4.1 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.ada` containing the following:

```ada
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.ada
gprbuild p_main
```

This generates source files for our project, extracted from `example_ada`, 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.ada`.

---

[7] 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 chapter[8] of the GNAT User's Guide.

4.2 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 section[9] 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:

```ada
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;
```

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:

---

[8] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#renaming-files-with-gnatchop

[9] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#conditional-compilation

```
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 chapter[10] 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).

4.3 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 this tool by linking our application to a special version of the memory allocation library (libgmem.a).

---

[10] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html# preprocessing-with-gnatprep

Let's consider this simple example:

```ada
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[11] of the GNAT User's Guide for a more detailed discussion of **gnatmem**.

4.4 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:

```
gnatdoc -P some_directory/default.gpr

# Alternatively, when the :file:`default.gpr` file is in the same directory
```

---

[11] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_and_program_execution.html#the-gnatmem-tool

```
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:

```ada
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;
```

```ada
package body P is

   function Add_One (V : Integer) return Integer is
   begin
      return V + 1;
   end Add_One;

end P;
```

```ada
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:

```ada
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

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[12].

4.5 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 aspects of the source and implementing many coding styles. These are extensively discussed in the section on gnatpp[13] of the GNAT User's Guide.

---

[12] http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/index.html
[13] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-pretty-printer-gnatpp

4.6 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:

```ada
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:

```ada
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.

This is a good starting point for the implementation of the body. Please refer to the section on gnatstub[14] of the GNAT User's Guide for a detailed discussion of **gnatstub** and its options.

---

[14] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-body-stub-generator-gnatstub