

GNAT User's Guide for Native Platforms

GNAT User's Guide for Native Platforms , May 19, 2026

AdaCore

Copyright © 2008-2026, Free Software Foundation

‘GNAT, The GNU Ada Development Environment’

GCC version 17.0.0

AdaCore

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being “GNAT User’s Guide for Native Platforms”, and with no Back-Cover Texts. A copy of the license is included in the section entitled [GNU Free Documentation License], page 318.

1 About This Guide

This guide describes the use of GNAT, a compiler and software development toolset for the full Ada programming language. It documents the features of the compiler and tools, and explains how to use them to build Ada applications.

GNAT implements Ada 95, Ada 2005, Ada 2012, and Ada 2022. You may also invoke it in Ada 83 compatibility mode. By default, GNAT assumes Ada 2012, but you can use a compiler switch ([Compiling Different Versions of Ada], page 145) to explicitly specify the language version. Throughout this manual, references to ‘Ada’ without a year suffix apply to all versions of the Ada language starting with Ada 95.

GNAT supports both the GCC and LLVM back end compilation families. Most GNAT versions use the GCC back end, but some are now available using the LLVM back end. In some places in this manual, we distinguish between the two back ends, but in most cases, everything in this manual applies to both back ends. We refer to GNAT with the LLVM back end as ‘GNAT LLVM’. See [GNAT with the LLVM Back End], page 176, for limitations of GNAT LLVM.

1.1 What This Guide Contains

This guide contains the following chapters:

- * [Getting Started with GNAT], page 3, describes how to get started compiling and running Ada programs with the GNAT Ada programming environment.
- * [The GNAT Compilation Model], page 6, describes the compilation model used by GNAT.
- * [Building Executable Programs with GNAT], page 76, describes how to use the main GNAT tools to build executable programs, and it also gives examples of using the GNU make utility with GNAT.
- * [GNAT Utility Programs], page 177, explains the various utility programs that are included in the GNAT environment.
- * [GNAT and Program Execution], page 183, covers a number of topics related to running, debugging, and tuning the performance of programs developed with GNAT.

Appendices cover several additional topics:

- * [Platform-Specific Information], page 241, describes the different run-time library implementations and also presents information on how to use GNAT on several specific platforms.
- * [Example of Binder Output File], page 270, shows the source code for the binder output file for a sample program.
- * [Elaboration Order Handling in GNAT], page 286, describes how GNAT helps you deal with elaboration order issues.
- * [Inline Assembler], page 308, shows how to use the inline assembly facility in an Ada program.

1.2 What You Should Know before Reading This Guide

This guide assumes a basic familiarity with the Ada 95 language, as described in the International Standard ANSI/ISO/IEC-8652:1995, January 1995. Reference manuals for Ada 95, Ada 2005, Ada 2012 and Ada 2022 are included in the GNAT documentation package.

1.3 Related Information

For further information about Ada and related tools, please refer to the following documents:

- * *Ada 95 Reference Manual*, *Ada 2005 Reference Manual*, *Ada 2012 Reference Manual*, and *Ada 2022 Reference Manual*, which contain reference material for the several revisions of the Ada language standard.
- * *GNAT Reference Manual*, which contains all reference material for the GNAT implementation of Ada.
- * *Using GNAT Studio*, which describes the GNAT Studio Integrated Development Environment.
- * *GNAT Studio Tutorial*, which introduces the main GNAT Studio features through examples.
- * *Debugging with GDB*, for all details on the use of the GNU source-level debugger.

1.4 Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- * Functions, utility program names, standard names, and classes.
- * Option flags
- * File names
- * Variables
- * ‘Emphasis’
- * [optional information or parameters]
- * Examples are described by text
and then shown this way.
- * Commands that you enter are shown as preceded by a prompt string comprising the \$ character followed by a space.
- * Full file names are shown with the ‘/’ character as the directory separator; e.g., `parent-dir/subdir/myfile.adb`. If you are using GNAT on a Windows platform, please note that you should use the ‘\’ character instead.

2 Getting Started with GNAT

This chapter describes how to use GNAT’s command line interface to build executable Ada programs. On most platforms a visually oriented Integrated Development Environment is also available: GNAT Studio. GNAT Studio offers a graphical “look and feel”, support for development in other programming languages, comprehensive browsing features, and many other capabilities. For information on GNAT Studio please refer to the *GNAT Studio documentation*.

2.1 System Requirements

Even though any machine can run the GNAT toolset and GNAT Studio IDE, to get the best experience we recommend using a machine with as many cores as possible, allowing individual compilations to run in parallel. A comfortable setup for a compiler server is a machine with 24 physical cores or more, with at least 48 GB of memory (2 GB per core).

For a desktop machine, we recommend a minimum of 4 cores (8 is preferred), with at least 2GB per core (so 8 to 16GB).

In addition, for running and smoothly navigating sources in GNAT Studio, we recommend at least 1.5 GB, plus 3 GB of RAM per million source lines of code. So we recommend at least 3 GB for 500K lines of code and 7.5 GB for 2 million lines of code.

Using fast, local drives can make a significant difference in build and link times. You should avoid network drives such as NFS, SMB, or worse, configuration management filesystems (such as ClearCase dynamic views) as much as possible since these will produce very degraded performance (typically 2 to 3 times slower than on fast, local drives). If you cannot avoid using such slow drives for accessing source code, you should at least configure your project file so the result of the compilation is stored on a drive local to the machine performing the compilation. You can do this by setting the `Object_Dir` project file attribute.

2.2 Running GNAT

You need to take three steps to create an executable file from an Ada source file:

- * You must compile the source file(s).
- * You must bind the file(s) using the GNAT binder.
- * You must link all appropriate object files to produce an executable.

You most commonly perform all three steps by using the `gnatmake` utility program. You pass it the name of the main program and it automatically performs the necessary compilation, binding, and linking steps.

2.3 Running a Simple Ada Program

You may use any text editor to prepare an Ada program. (If you use Emacs, an optional Ada mode may be helpful in laying out the program.) The program text is a normal text file. We will assume in our initial example that you have used your editor to prepare the following standard format text file named `hello.adb`:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
```

```
begin
  Put_Line ("Hello WORLD!");
end Hello;
```

With the normal default file naming conventions, GNAT requires that each file contain a single compilation unit whose file name is the unit name with periods replaced by hyphens; the extension is **ads** for a spec and **adb** for a body. You can override this default file naming convention by use of the special pragma **Source_File_Name** (see [Using Other File Names], page 12). Alternatively, if you want to rename your files according to this default convention, which is probably more convenient if you will be using GNAT for all your compilations, then you can use the **gnatchop** utility to generate correctly-named source files (see [Renaming Files with gnatchop], page 20).

You can compile the program using the following command (\$ is used as the command prompt in the examples in this document):

```
$ gcc -c hello.adb
```

gcc is the command used to run the compiler. It is capable of compiling programs in several languages, including Ada and C. It assumes you have given it an Ada program if the file extension is either **.ads** or **.adb**, in which case it will call the GNAT compiler to compile the specified file.

The **-c** switch is required. It tells **gcc** to only do a compilation. (For C programs, **gcc** can also do linking, but this capability is not used directly for Ada programs, so you must always specify the **-c**.)

This compile command generates a file **hello.o**, which is the object file corresponding to your Ada program. It also generates an ‘Ada Library Information’ file **hello.ali**, which contains additional information used to check that an Ada program is consistent.

To build an executable file, use either **gnatmake** or **gprbuild** with the name of the main file: these tools are builders that perform all the necessary build steps in the correct order. In particular, these builders automatically recompile any sources that have been modified since they were last compiled, as well as sources that depend on such modified sources, so that ‘version skew’ is avoided.

```
$ gnatmake hello.adb
```

The result is an executable program called **hello**, which you can run by entering:

```
$ hello
```

assuming that the current directory is on the search path for executable programs.

and, if all has gone well, you will see:

```
Hello WORLD!
```

appear in response to this command.

2.4 Running a Program with Multiple Units

Consider a slightly more complicated example with three files: a main program and the spec and body of a package:

```
package Greetings is
  procedure Hello;
  procedure Goodbye;
```

```

end Greetings;

with Ada.Text_IO; use Ada.Text_IO;
package body Greetings is
  procedure Hello is
  begin
    Put_Line ("Hello WORLD!");
  end Hello;

  procedure Goodbye is
  begin
    Put_Line ("Goodbye WORLD!");
  end Goodbye;
end Greetings;

with Greetings;
procedure Gmain is
begin
  Greetings.Hello;
  Greetings.Goodbye;
end Gmain;

```

Following the one-unit-per-file rule, place this program in the following three separate files:

```

'greetings.ads'
    spec of package Greetings

'greetings.adb'
    body of package Greetings

'gmain.adb'
    body of main program

```

Note that there is no required order of compilation when using GNAT. In particular it is perfectly fine to compile the main program first. Also, it is not necessary to compile package specs in the case where there is an accompanying body; you only need compile the body. If you want to submit these files to the compiler for semantic checking and not code generation, use the `-gnatc` switch:

```
$ gcc -c greetings.ads -gnatc
```

Although you can do the compilation in separate steps, in practice it's almost always more convenient to use the `gnatmake` or `gprbuild` tools:

```
$ gnatmake gmain.adb
```

3 The GNAT Compilation Model

This chapter describes the compilation model used by GNAT. Although similar to that used by other languages such as C and C++, this model is substantially different from the traditional Ada compilation models, which are based on a centralized program library. The chapter covers the following material:

- * Topics related to source file makeup and naming
 - * [Source Representation], page 7,
 - * [Foreign Language Representation], page 8,
 - * [File Naming Topics and Utilities], page 11,
- * [Configuration Pragmas], page 25,
- * [Generating Object Files], page 28,
- * [Source Dependencies], page 28,
- * [The Ada Library Information Files], page 29,
- * [Binding an Ada Program], page 30,
- * [GNAT and Libraries], page 30,
- * [Conditional Compilation], page 39,
- * [Mixed Language Programming], page 51,
- * [GNAT and Other Compilation Models], page 74,
- * [Using GNAT Files with External Tools], page 75,

3.1 Source Representation

Ada source programs are represented in standard text files, using Latin-1 coding. Latin-1 is an 8-bit code that includes the familiar 7-bit ASCII set plus additional characters used for representing foreign languages (see [Foreign Language Representation], page 8, for support of non-USA character sets). The format effector characters are represented using their standard ASCII encodings, as follows:

Character	Effect	Code
VT	Vertical tab	16#0B#
HT	Horizontal tab	16#09#
CR	Carriage return	16#0D#
LF	Line feed	16#0A#
FF	Form feed	16#0C#

Source files are in standard text file format. In addition, GNAT recognizes a wide variety of stream formats, in which the end of physical lines is marked by any of the following sequences: LF, CR, CR-LF, or LF-CR. This is useful in accommodating files imported from other operating systems.

Latin-1 letters, but these letters do not have the same encoding as Latin-1. In this mode, these letters are allowed in identifiers with uppercase and lowercase equivalence.

‘IBM PC (code page 850)’

This code page is a modification of 437 extended to include all the Latin-1 letters, but still not with the usual Latin-1 encoding. In this mode, all these letters are allowed in identifiers with uppercase and lowercase equivalence.

‘Full Upper 8-bit’

Any character in the range 80-FF is allowed in identifiers and all are considered distinct. In other words, there are no uppercase and lowercase equivalences in this range. This is useful in conjunction with certain encoding schemes used for some foreign character sets (e.g., the typical method of representing Chinese characters on the PC).

‘No Upper-Half’

No upper-half characters in the range 80-FF are allowed in identifiers. This gives Ada 83 compatibility for identifier names.

For precise data on the encodings permitted, and the uppercase and lowercase equivalences that are recognized, see the file `csets.adb` in the GNAT compiler sources. You will need to obtain a full source release of GNAT to obtain this file.

3.2.3 Wide_Character Encodings

GNAT allows wide character codes to appear in character and string literals, and also optionally in identifiers, by means of the following possible encoding schemes:

‘Hex Coding’

In this encoding, a wide character is represented by the following five character sequence:

ESC a b c d

where `a`, `b`, `c`, `d` are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, ESC A345 is used to represent the wide character with code `16#A345#`. This scheme is compatible with use of the full Wide_Character set.

‘Upper-Half Coding’

The wide character with encoding `16#abcd#` where the upper bit is on (in other words, ‘a’ is in the range 8-F) is represented as two bytes, `16#ab#` and `16#cd#`. The second byte cannot be a format control character, but is not required to be in the upper half. This method can be also used for shift-JIS or EUC, where the internal coding matches the external coding.

‘Shift JIS Coding’

A wide character is represented by a two-character sequence, `16#ab#` and `16#cd#`, with the restrictions described for upper-half encoding as described above. The internal character code is the corresponding JIS character according to the standard algorithm for Shift-JIS conversion. You can only use characters defined in the JIS code set table with this encoding method.

‘EUC Coding’

A wide character is represented by a two-character sequence `16#ab#` and `16#cd#`, with both characters being in the upper half. The internal character code is the corresponding JIS character according to the EUC encoding algorithm. You can only use characters defined in the JIS code set table with this encoding method.

‘UTF-8 Coding’

A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value, the representation is a one, two, or three byte sequence:

```
16#0000#-16#007f#: 2#0xxxxxxx#
16#0080#-16#07ff#: 2#110xxxxx# 2#10xxxxxx#
16#0800#-16#ffff#: 2#1110xxxx# 2#10xxxxxx# 2#10xxxxxx#
```

where the `xxx` bits correspond to the left-padded bits of the 16-bit character value. Note that all lower half ASCII characters are represented as ASCII bytes and all upper half characters and other wide characters are represented as sequences of upper-half (The full UTF-8 scheme allows for encoding 31-bit characters as 6-byte sequences the use of these sequences is documented in the following section on wide wide characters.)

‘Brackets Coding’

In this encoding, a wide character is represented by the following eight character sequence:

```
[ " a b c d " ]
```

where `a`, `b`, `c`, `d` are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, `['A345']` is used to represent the wide character with code `16#A345#`. You can also (though you are not required to) use the Brackets coding for upper half characters. For example, you can represent the code `16#A3#` as `['A3']`.

This scheme is compatible with use of the full `Wide_Character` set, and is also the method used for wide character encoding in some standard ACATS (Ada Conformity Assessment Test Suite) test suite distributions.

Note: Some of these coding schemes do not permit the full use of the Ada character set. For example, neither Shift JIS nor EUC allow the use of the upper half of the Latin-1 set.

3.2.4 Wide_Wide_Character Encodings

GNAT allows wide wide character codes to appear in character and string literals, and also optionally in identifiers, by means of the following possible encoding schemes:

‘UTF-8 Coding’

A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value,

the representation of character codes with values greater than 16#FFFF# is a four, five, or six byte sequence:

```

16#01_0000#-16#10_FFFF#: 11110xxx 10xxxxxx 10xxxxxx
                          10xxxxxx
16#0020_0000#-16#03FF_FFFF#: 111110xx 10xxxxxx 10xxxxxx
                              10xxxxxx 10xxxxxx
16#0400_0000#-16#7FFF_FFFF#: 1111110x 10xxxxxx 10xxxxxx
                              10xxxxxx 10xxxxxx 10xxxxxx

```

where the xxx bits correspond to the left-padded bits of the 32-bit character value.

‘Brackets Coding’

In this encoding, a wide wide character is represented by the following ten or twelve byte character sequence:

```

[ " a b c d e f " ]
[ " a b c d e f g h " ]

```

where a-h are the six or eight hexadecimal characters (using uppercase letters) of the wide wide character code. For example, ["1F4567"] is used to represent the wide wide character with code 16#001F_4567#.

This scheme is compatible with use of the full `Wide_Wide_Character` set, and is also the method used for wide wide character encoding in some standard ACATS (Ada Conformity Assessment Test Suite) test suite distributions.

3.3 File Naming Topics and Utilities

GNAT has a default file naming scheme, but also provides you with a high degree of control over how the names and extensions of your source files correspond to the Ada compilation units that they contain.

3.3.1 File Naming Rules

GNAT determines the default file name by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit, replacing the separating dots with hyphens, and using lowercase for all letters.

An exception occurs if the file name generated by the above rules starts with one of the characters a, g, i, or s and the second character is a hyphen. In this case, the character tilde is used in place of the hyphen. This special rule avoids clashes with the standard names for child units of the packages `System`, `Ada`, `Interfaces`, and `GNAT`, which use the prefixes s-, a-, i-, and g-, respectively.

The file extension is `.ads` for a spec and `.adb` for a body. The following table shows some examples of these rules.

Source File	Ada Compilation Unit
<code>main.ads</code>	Main (spec)
<code>main.adb</code>	Main (body)

<code>arith_functions.ads</code>	Arith_Functions (package spec)
<code>arith_functions.adb</code>	Arith_Functions (package body)
<code>func-spec.ads</code>	Func.Spec (child package spec)
<code>func-spec.adb</code>	Func.Spec (child package body)
<code>main-sub.adb</code>	Sub (subunit of Main)
<code>a~bad.adb</code>	A.Bad (child package body)

Following these rules can result in excessively long file names if corresponding unit names are long (for example, if child units or subunits are heavily nested). An option is available to shorten such long file names (called file name ‘krunching’). You may find this particularly useful when programs being developed with GNAT are to be used on operating systems with limited file name lengths. [Using gnatkr], page 18.

Of course, no file shortening algorithm can guarantee uniqueness over all possible unit names; if file name krunching is used, it is your responsibility to ensure no name clashes occur. Alternatively, you can specify the exact file names that you want used, as described in the next section. Finally, if your Ada programs are migrating from a compiler with a different naming convention, you can use the `gnatchop` utility to produce source files that follow the GNAT naming conventions. (For details see [Renaming Files with gnatchop], page 20.)

Note: in the case of Windows or Mac OS operating systems, case is not significant. So, for example, on Windows if the canonical name is `main-sub.adb`, you can use the file name `Main-Sub.adb` instead. However, case is significant for other operating systems, so, for example, if you want to use other than canonically cased file names on a Unix system, you need to follow the procedures described in the next section.

3.3.2 Using Other File Names

The previous section described the default rules used by GNAT to determine the file name in which a given unit resides. It is usually convenient to follow these default rules, and if you follow them, the compiler knows without being explicitly told where to find all the files it needs.

However, in some cases, particularly when a program is imported from another Ada compiler environment, it may be more convenient for you to specify which file names contain which units. GNAT allows arbitrary file names to be used by means of the `Source_File_Name` pragma. The form of this pragma is as shown in the following examples:

```
pragma Source_File_Name (My_Uutilities.Stacks,
    Spec_File_Name => "myutilst_a.adb");
pragma Source_File_Name (My_Uutilities.Stacks,
    Body_File_Name => "myutilst.adb");
```

As shown in this example, the first argument for the pragma is the unit name (in this example a child unit). The second argument has the form of a named association. The

identifier indicates whether the file name is for a spec or a body; the file name itself is given by a string literal.

The source file name pragma is a configuration pragma, which means that normally you will place it in the `gnat.adc` file used to hold configuration pragmas that apply to a complete compilation environment. For more details on how the `gnat.adc` file is created and used see [Handling of Configuration Pragmas], page 26.

GNAT allows you to specify completely arbitrary file names using the source file name pragma. However, if the file name specified has an extension other than `.ads` or `.adb` you must use a special syntax when compiling the file. The name on the command line in this case must be preceded by the special sequence `-x` followed by a space and the name of the language, here `ada`, as in:

```
$ gcc -c -x ada peculiar_file_name.sim
```

`gnatmake` handles non-standard file names in the usual manner (the non-standard file name for the main program is simply used as the argument to `gnatmake`). Note that if the extension is also non-standard, you must include it in the `gnatmake` command; it may not be omitted.

3.3.3 Alternative File Naming Schemes

The previous section described the use of the `Source_File_Name` pragma to allow arbitrary names to be assigned to individual source files. However, this approach requires one pragma for each file and, especially in large systems, can result in very long `gnat.adc` files, which can create a maintenance problem.

GNAT also provides a facility for specifying systematic file naming schemes other than the standard default naming scheme previously described. An alternative scheme for naming is specified by the use of `Source_File_Name` pragmas having the following format:

```
pragma Source_File_Name (
  Spec_File_Name   => FILE_NAME_PATTERN
  [ , Casing        => CASING_SPEC ]
  [ , Dot_Replacement => STRING_LITERAL ] );

pragma Source_File_Name (
  Body_File_Name   => FILE_NAME_PATTERN
  [ , Casing        => CASING_SPEC ]
  [ , Dot_Replacement => STRING_LITERAL ] );

pragma Source_File_Name (
  Subunit_File_Name => FILE_NAME_PATTERN
  [ , Casing        => CASING_SPEC ]
  [ , Dot_Replacement => STRING_LITERAL ] );

FILE_NAME_PATTERN ::= STRING_LITERAL
CASING_SPEC ::= Lowercase | Uppercase | Mixedcase
```

The `FILE_NAME_PATTERN` string shows how the file name is constructed. It contains a single asterisk character, and the unit name is substituted systematically for this asterisk. The optional parameter `Casing` indicates whether the unit name is to be all upper-case letters,

all lower-case letters, or mixed-case. If no `Casing` parameter is used, the default is all lower-case.

You use the optional `Dot_Replacement` string to replace any periods that occur in subunit or child unit names. If you don't specify a `Dot_Replacement` argument, separating dots appear unchanged in the resulting file name. The above syntax indicates that the `Casing` argument must appear before the `Dot_Replacement` argument, but you can write these arguments in any order.

As indicated, you can specify different naming schemes for bodies, specs, and subunits. Quite often, the rule for subunits is the same as the rule for bodies, in which case, you need not provide a separate `Subunit_File_Name` rule; in this case the `Body_File_Name` rule is used for subunits as well.

You can also use the separate rule for subunits to implement the rather unusual case of a compilation environment (e.g., a single directory) which contains a subunit and a child unit with the same unit name. Although both units cannot appear in the same partition, the Ada Reference Manual allows (but does not require) the possibility of the two units coexisting in the same environment.

File name translation consists of the following steps:

- * If there is a specific `Source_File_Name` pragma for the given unit, this is always used and any general pattern rules are ignored.
- * If there is a pattern type `Source_File_Name` pragma that applies to the unit, the resulting file name is used if the file exists. If more than one pattern matches, the latest one is tried first and the first attempt that results in a reference to a file that exists is used.
- * If no pattern type `Source_File_Name` pragma that applies to the unit for which the corresponding file exists, the standard GNAT default naming rules are used.

As an example of the use of this mechanism, consider a commonly used scheme in which file names are all lower case, with separating periods copied unchanged to the resulting file name, specs end with `.1.adb`, and bodies end with `.2.adb`. GNAT will follow this scheme if the following two pragmas appear:

```
pragma Source_File_Name
  (Spec_File_Name => ".1.adb");
pragma Source_File_Name
  (Body_File_Name => ".2.adb");
```

The default GNAT scheme is equivalent to providing the following default pragmas:

```
pragma Source_File_Name
  (Spec_File_Name => ".ads", Dot_Replacement => "-");
pragma Source_File_Name
  (Body_File_Name => ".adb", Dot_Replacement => "-");
```

Our final example implements a scheme typically used with one of the legacy Ada 83 compilers, where the separator character for subunits was `'_'` (two underscores), specs were identified by adding `_ADA`, bodies by adding `.ADA`, and subunits by adding `.SEP`. All file names were upper case. Child units were not present, of course, since this was an Ada 83 compiler, but it seems reasonable to extend this scheme to use the same double underscore separator for child units.

```
pragma Source_File_Name
  (Spec_File_Name => ".ADA",
   Dot_Replacement => "__",
   Casing = Uppercase);
pragma Source_File_Name
  (Body_File_Name => ".ADA",
   Dot_Replacement => "__",
   Casing = Uppercase);
pragma Source_File_Name
  (Subunit_File_Name => ".SEP",
   Dot_Replacement => "__",
   Casing = Uppercase);
```

3.3.4 Handling Arbitrary File Naming Conventions with `gnatname`

3.3.4.1 Arbitrary File Naming Conventions

The GNAT compiler must know the source file name of a compilation unit in order to compile it. When using the standard GNAT default file naming conventions (`.ads` for specs, `.adb` for bodies), it does not need additional information.

When the source file names do not follow the standard GNAT default file naming conventions, you must give the GNAT compiler additional information through a configuration pragmas file ([Configuration Pragmas], page 25) or a project file. When the non-standard file naming conventions are well-defined, a small number of pragmas `Source_File_Name` specifying a naming pattern ([Alternative File Naming Schemes], page 13) may be sufficient. However, if the file naming conventions are irregular or arbitrary, you must define a number of pragma `Source_File_Name` for individual compilation units. To help maintain the correspondence between compilation unit names and source file names within the compiler, GNAT provides a tool `gnatname` to generate the required pragmas for a set of files.

3.3.4.2 Running `gnatname`

The usual form of the `gnatname` command is:

```
$ gnatname [ switches ] naming_pattern [ naming_patterns ]
  [--and [ switches ] naming_pattern [ naming_patterns ]]
```

All of the arguments are optional. If invoked without any arguments, `gnatname` will display its usage.

When used with at least one naming pattern, `gnatname` attempts to find all the compilation units in files that follow at least one of the naming patterns. To find these compilation units, `gnatname` uses the GNAT compiler in syntax-check-only mode on all regular files.

One or several ‘Naming Patterns’ may be given as arguments to `gnatname`. Each Naming Pattern is enclosed between double quotes (or single quotes on Windows). A Naming Pattern is a regular expression similar to the wildcard patterns used in file names by the Unix shells or the DOS prompt.

You may call `gnatname` with several sections of directories/patterns. Sections are separated by the switch `--and`. In each section, you must include at least one pattern. If you don’t

specify a directory a section, the current directory (or the project directory if `-P` is used) is used. The options other than the directory switches and the patterns apply globally even if they are in different sections.

Examples of Naming Patterns are:

```
"*. [12] .ada"
"*.[a-z][sb]*"
"body_*"      "spec_*
```

For a more complete description of the syntax of Naming Patterns, see the second kind of regular expressions described in `g-regex.ads` (the ‘Glob’ regular expressions).

When invoked without the switch `-P`, `gnatname` will create a configuration pragmas file `gnat.adc` in the current working directory, with pragmas `Source_File_Name` for each file that contains a valid Ada unit.

3.3.4.3 Switches for `gnatname`

Switches for `gnatname` must precede any specified Naming Pattern.

You may specify any of the following switches to `gnatname`:

- `--version`
Display Copyright and version, then exit disregarding, all other options.
- `--help`
If `--version` was not used, display usage, then exit, disregarding all other options.
- `--subdirs='dir'`
Actual object, library or exec directories are subdirectories `<dir>` of the specified ones.
- `--no-backup`
Do not create a backup copy of an existing project file.
- `--and`
Start another section of directories/patterns.
- `-c`filename'`
Create a configuration pragmas file `filename` (instead of the default `gnat.adc`). There may be zero, one, or more space between `-c` and `filename`. `filename` may include directory information. `filename` must be writable. You can specify only one switch `-c`. When a switch `-c` is specified, you may not specify switch `-P` (see below).
- `-d`dir'`
Look for source files in directory `dir`. You may put zero, one or more spaces between `-d` and `dir`. `dir` may end with `/**`, i.e., you may write it the form `root_dir/**`. In this case, the directory `root_dir` and all of its subdirectories, recursively, have to be searched for sources. When you specify a `-d` switch, the current working directory will not be searched for source files unless you explicitly specify it with a `-d` or `-D` switch. You may specify several switches `-d`. If `dir` is a relative path, it is relative to the directory of the configuration

in the output. For example, when krunching `hellofile.ads` to eight characters, the result will be `hellofil.ads`.

Note: for compatibility with previous versions of `gnatkr`, you can use dots in the name instead of hyphens, but `gnatkr` always interprets the last dot as the start of an extension. So if you pass `gnatkr` an argument such as `Hello.World.adb`, it treats it exactly as if the first period had been a hyphen, so, for example, krunching to eight characters gives the result `hellworl.adb`.

Note that the result is always all lower case. Other characters are folded as required.

`length` represents the length of the krunched name. The default if you don't specify it, is 8 characters. A length of zero means unlimited, in other words don't chop except for system files where the implied krunching length is always eight characters.

The output is the krunched name. The output has an extension only if the original argument was a file name with an extension.

3.3.5.3 Krunching Method

The initial file name is determined by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit and replacing the separating dots with hyphens and using lowercase for all letters, except that a hyphen in the second character position is replaced by a tilde if the first character is `a`, `i`, `g`, or `s`. The extension is `.ads` for a spec and `.adb` for a body. Krunching does not affect the extension, but the file name is shortened to the specified length by following these rules:

- * The name is divided into segments separated by hyphens, tildes, or underscores and all hyphens, tildes, and underscores are eliminated. If this leaves the name short enough, we are done.
- * If the name is too long, the longest segment is located (left-most if there are two of equal length) and shortened by dropping its last character. This is repeated until the name is short enough.

As an example, consider the krunching of `our-strings-wide_fixed.adb` to fit the name into 8 characters, as required by some operating systems:

```
our-strings-wide_fixed 22
our strings wide fixed 19
our string  wide fixed 18
our strin  wide fixed 17
our stri   wide fixed 16
our stri   wide fixe  15
our str     wide fixe  14
our str     wid  fixe  13
our str     wid  fix   12
ou  str     wid  fix   11
ou  st      wid  fix   10
ou  st      wi   fix    9
ou  st      wi   fi     8
Final file name: oustwifi.adb
```

- * The file names for all predefined units are always krunched to eight characters. The krunching of these predefined units uses the following special prefix replacements:

Prefix	Replacement
<code>ada-</code>	<code>a-</code>
<code>gnat-</code>	<code>g-</code>
<code>interfac es-</code>	<code>i-</code>
<code>system-</code>	<code>s-</code>

These system files have a hyphen in the second character position. That's is why normal user files replace such a character with a tilde.

As an example of this special rule, consider `ada-strings-wide_fixed.adb`, which gets krunched as follows:

```
ada-strings-wide_fixed 22
a- strings wide fixed 18
a- string  wide fixed 17
a- strin   wide fixed 16
a- stri    wide fixed 15
a- stri    wide fixe  14
a- str     wide fixe  13
a- str     wid  fixe  12
a- str     wid  fix   11
a- st      wid  fix   10
a- st      wi   fix    9
a- st      wi   fi     8
Final file name: a-stwifi.adb
```

Of course, no file shortening algorithm can guarantee uniqueness over all possible unit names. If file name krunching is used, it's your responsibility to ensure that no name clashes occur. The utility program `gnatkr` is supplied so that you can conveniently determine the krunched name of a file.

3.3.5.4 Examples of gnatkr Usage

```
$ gnatkr very_long_unit_name.ads --> velounna.ads
$ gnatkr grandparent-parent-child.ads --> grparchi.ads
$ gnatkr Grandparent.Parent.Child.ads --> grparchi.ads
$ gnatkr grandparent-parent-child --> grparchi
$ gnatkr very_long_unit_name.ads/count=6 --> vlunna.ads
$ gnatkr very_long_unit_name.ads/count=0 --> very_long_unit_name.ads
```

3.3.6 Renaming Files with gnat Chop

This section discusses how to handle files with multiple units by using the `gnat Chop` utility. You will also find this utility useful in renaming files to meet the standard GNAT default file naming conventions.

3.3.6.1 Handling Files with Multiple Units

GNAT's fundamental compilation model requires that a file submitted to the compiler contain only one unit and there be a strict correspondence between the file name and the unit name.

If you want to have your files contain multiple units, perhaps to maintain compatibility with some other Ada compilation system, you can use **gnatname** to generate or update your project files, which can be processed by GNAT.

See [Handling Arbitrary File Naming Conventions with **gnatname**], page 15, for more details on how to use *gnatname*.

Alternatively, if you want to permanently restructure a set of 'foreign' files so that they match the GNAT rules, and do the remaining development using the GNAT structure, you can simply use **gnatchop** once, generate the new set of files containing only one unit per file, and work with them from that point on.

Note that if your file containing multiple units starts with a byte order mark (BOM) specifying UTF-8 encoding, each file generated by **gnatchop** will start with a copy of this BOM, meaning that they can be compiled automatically in UTF-8 mode without you needing to specify an explicit encoding.

3.3.6.2 Operating **gnatchop** in Compilation Mode

The basic function of **gnatchop** is to take a file with multiple units and split it into separate files. The boundary between units is reasonably clear, except for the issue of comments and pragmas. In default mode, the rule is that any pragmas between units belong to the previous unit, except that configuration pragmas always belong to the following unit. Any comments belong to the following unit. These rules almost always result in the right choice of the split point without you needing to mark it explicitly and you'll likely find this default to be what you want. In this default mode, you may not submit a file containing only configuration pragmas, or one that ends in configuration pragmas, to **gnatchop**.

However, using a special switch to activate 'compilation mode', **gnatchop** can perform another function, which is to provide exactly the semantics required by the RM for the handling of configuration pragmas in a compilation. In the absence of configuration pragmas at the main file level, this switch has no effect, but it causes such configuration pragmas to be handled in a very different manner.

First, in compilation mode, if you give **gnatchop** a file that consists of only configuration pragmas, it appends this file to the **gnat.adc** file in the current directory. This behavior provides the required behavior described in the RM for the actions to be taken on submitting such a file to the compiler, namely that these pragmas should apply to all subsequent compilations in the same compilation environment. Using GNAT, the current directory, possibly containing a **gnat.adc** file is the representation of a compilation environment. For more information on the **gnat.adc** file, see [Handling of Configuration Pragmas], page 26.

Second, in compilation mode, if you give **gnatchop** a file that starts with configuration pragmas and contains one or more units, then configuration pragmas are prepended to each of the chopped files. This behavior provides the required behavior described in the RM for the actions to be taken on compiling such a file, namely that the pragmas apply to all units in the compilation, but not to subsequently compiled units.

Finally, if configuration pragmas appear between units, they are appended to the previous unit. This results in the previous unit being illegal, since the compiler does not accept configuration pragmas that follow a unit. This provides the required RM behavior that forbids configuration pragmas other than those preceding the first compilation unit of a compilation.

For most purposes, you will use **gnatchop** in default mode. You only use the compilation mode described above if you need precisely accurate behavior with respect to compilations and you have files that contain multiple units and configuration pragmas. In this circumstance, the use of **gnatchop** with the compilation mode switch provides the required behavior. This is the mode in which GNAT processes the ACVC tests.

3.3.6.3 Command Line for **gnatchop**

You call **gnatchop** as follows:

```
$ gnatchop switches file_name [file_name ...]
    [directory]
```

The only required argument is the file name of the file to be chopped. There are no restrictions on the form of this file name. The file itself contains one or more Ada units, in normal GNAT format, concatenated together. As shown, more than one file may be presented to be chopped.

When run in default mode, **gnatchop** generates one output file in the current directory for each unit in each of the files.

directory, if specified, gives the name of the directory to which the output files will be written. If you don't specify it, all files are written to the current directory.

For example, given a file called **hellofiles** containing

```
procedure Hello;

with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
    Put_Line ("Hello");
end Hello;
```

the command

```
$ gnatchop hellofiles
```

generates two files in the current directory, one called **hello.ads** containing the single line that is the procedure spec, and the other called **hello.adb** containing the remaining text. The original file is not affected. You can compile these generated files in the normal manner.

When you invoke **gnatchop** on a file that is empty or contains only empty lines and/or comments, **gnatchop** will complete normally, but won't produce any new file.

For example, given a file called **toto.txt** containing

```
-- Just a comment
```

the command

```
$ gnatchop toto.txt
```


use the `-w` switch, in which case the last occurrence in the last file will be the one that is output and `gnatchop` will skip earlier duplicate occurrences for the same unit.

3.4 Configuration Pragmas

Configuration pragmas supported by GNAT consist of those pragmas described as such in the Ada Reference Manual and the implementation-dependent pragmas that are configuration pragmas. See the `Implementation_Defined_Pragmas` chapter in the *GNAT-Reference-Manual* for details on these additional GNAT-specific configuration pragmas. Most notably, the pragma `Source_File_Name`, which allows specifying non-default names for source files, is a configuration pragma. The following is a complete list of configuration pragmas recognized by GNAT:

```
Ada_83
Ada_95
Ada_05
Ada_2005
Ada_12
Ada_2012
Ada_2022
Aggregate_Individually_Assign
Allow_Integer_Address
Annotate
Assertion_Policy
Assume_No_Invalid_Values
C_Pass_By_Copy
Check_Float_Overflow
Check_Name
Check_Policy
Component_Alignment
Convention_Identifier
Debug_Policy
Default_Scalar_Storage_Order
Default_Storage_Pool
Detect_Blocking
Disable_Atomic_Synchronization
Discard_Names
Elaboration_Checks
Eliminate
Enable_Atomic_Synchronization
Extend_System
Extensions_Allowed
External_Name_Casing
Fast_Math
Favor_Top_Level
Ignore_Pragma
Implicit_Packing
InitializeScalars
```

Interrupt_State
Interrupts_System_By_Default
License
Locking_Policy
No_Component_Reordering
No_Heap_Finalization
No_Strict_Aliasing
NormalizeScalars
Optimize_Alignment
Overflow_Mode
Overriding_Renamings
Partition_Elaboration_Policy
Persistent_BSS
Prefix_Exception_Messages
Priority_Specific_Dispatching
Profile
Profile_Warnings
Queuing_Policy
Rename_Pragma
Restrictions
Restriction_Warnings
Reviewable
Short_Circuit_And_Or
Source_File_Name
Source_File_Name_Project
SPARK_Mode
Style_Checks
Suppress
Suppress_Exception_Locations
Task_Dispatching_Policy
Unevaluated_Use_Of_Old
Unsuppress
Use_VADS_Size
User_Aspect_Definition
Validity_Checks
Warning_As_Error
Warnings
Wide_Character_Encoding

3.4.1 Handling of Configuration Pragmas

You can place configuration pragmas either appear at the start of a compilation unit or in a configuration pragma file that applies to all compilations performed in a given compilation environment.

Configuration pragmas placed before a library level package specification are not propagated to the corresponding package body (see RM 10.1.5(8)); they must be added explicitly to the package body.

If you are using project file, they provide a separate mechanism using project attributes.

3.5 Generating Object Files

An Ada program consists of a set of source files and the first step in compiling the program is generating the corresponding object files. You generate these by compiling a subset of these source files. The files you need to compile are the following:

- * If a package spec has no body, compile the package spec to produce the object file for the package.
- * If a package has both a spec and a body, compile the body to produce the object file for the package. You need not compile the source file for the package spec in this case because there's only one object file, which contains the code for both the spec and body of the package.
- * For a subprogram, compile the subprogram body to produce the object file for the subprogram. You need not compile the spec, if such a file is present.
- * In the case of subunits, only compile the parent unit. GNAT generates a single object file for the entire subunit tree, which includes all the subunits.
- * Compile child units independently of their parent units (though, of course, the spec of all the ancestor unit must be present in order to compile a child unit).
- * Compile generic units in the same manner as any other units. The object files in this case are small dummy files that contain, at most, the flag used for elaboration checking. This is because GNAT always handles generic instantiation by means of macro expansion. However, you still must compile generic units for dependency checking and elaboration purposes.

The preceding rules describe the set of files that must be compiled to generate all the object files for a program. See the following section on dependencies for more details on computing that set of files. Each object file has the same name as the corresponding source file, except that the extension is `.o`, as usual.

You may wish to compile other files for the purpose of checking their syntactic and semantic correctness. For example, in the case where a package has a separate spec and body, you would not normally compile the spec. However, it is convenient in practice to compile the spec to make sure it is error-free before compiling clients of this spec because such compilations will fail if there is an error in the spec.

GNAT provides an option for compiling such files purely for the purposes of checking correctness; such compilations are not required as part of the process of building a program. To compile a file in this checking mode, use the `-gnatc` switch.

3.6 Source Dependencies

Each object file obviously depends on at least the source file which is compiled to produce it. Here we are using “depends” in the sense of a typical `make` utility; in other words, an object file depends on a source file if changes to the source file require the object file to be recompiled. In addition to this basic dependency, a given object may depend on additional source files as follows:

- * If a file being compiled ‘with’s a unit `X`, the object file depends on the file containing the spec of unit `X`. This includes files that are ‘with’ed implicitly either because they are

- * Main program information (including priority and time slice settings, as well as the wide character encoding used during compilation).
- * List of arguments used in the compilation command
- * Attributes of the unit, including the configuration pragmas used, an indication of whether the compilation was successful, and the exception model used.
- * A list of relevant restrictions applying to the unit (used for consistency checking).
- * Categorization information (e.g., use of pragma `Pure`).
- * Information on all ‘with’ed units, including presence of `Elaborate` or `Elaborate_All` pragmas.
- * Information from any `Linker_Options` pragmas used in the unit
- * Information on the use of `Body_Version` or `Version` attributes in the unit.
- * Dependency information. This is a list of files, together with time stamp and checksum information. These are files on which the unit depends in the sense that the modification of any of these units requires the recompilation of the unit in question.
- * Cross-reference data. Contains information on all entities referenced in the unit. Used by some tools to provide cross-reference information.

For a full detailed description of the format of the ALI file, see the source of the spec of unit `Lib.Writ`, contained in file `lib-writ.ads` in the GNAT compiler sources.

3.8 Binding an Ada Program

When using languages such as C and C++, once the source files have been compiled the only remaining step in building an executable program is linking the object modules together. This means that you can link an inconsistent version of a program, in which two units have included different versions of the same header.

The rules of Ada do not permit such an inconsistent program to be built. For example, if two clients have different versions of the same package, it is illegal to build a program containing these two clients. These rules are enforced by the GNAT binder, which also determines an elaboration order consistent with the Ada rules.

The GNAT binder is run after all the object files for a program have been created. It is given the name of the main program unit and from this determines the set of units required by the program by reading the corresponding ALI files. It generates error messages if the program is inconsistent or if no valid order of elaboration exists.

If no errors are detected, the binder produces a main program in Ada that contains calls to the elaboration procedures of those compilation unit that require them, followed by a call to the main program. This Ada program is compiled to generate the object file for the main program. The name of the Ada file is `b~xxx.adb` (with the corresponding spec `b~xxx.ads`) where `xxx` is the name of the main program unit.

Finally, the linker is used to build the resulting executable program, using the object from the main program from the bind step as well as the object files for the Ada units of the program.


```

{
    t->method1 ();
}

int main ()
{
    A obj;
    adainit ();
    obj.method2 (3030);
    adafinal ();
}

//ex7.h

class Origin {
public:
    int o_value;
};
class A : public Origin {
public:
    void method1 (void);
    void method2 (int v);
    A();
    int  a_value;
};

//ex7.C

#include "ex7.h"
#include <stdio.h>

extern "C" { void ada_method2 (A *t, int v);}

void A::method1 (void)
{
    a_value = 2020;
    printf ("in A::method1, a_value = %d \\n",a_value);
}

void A::method2 (int v)
{
    ada_method2 (this, v);
    printf ("in A::method2, a_value = %d \\n",a_value);
}

A::A(void)
{
    a_value = 1010;
}

```


