\

GNAT, The GNU Ada 95 Compiler
Document revision level 1.122 $
GNAT Version 3.11p
August 30, 1998

Ada Core Technologies, Inc.

About This Guide

This guide describes the use of GNAT, the full language compiler for the Ada 95 programming language. It describes the features of the compiler, and details how you can use it to build Ada 95 applications.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Getting Started With GNAT], page 3, describes how to get started compiling and running Ada programs with GNAT, the Ada programming environment.
- Chapter 2 [The GNAT Compilation Model], page 7, describes the compilation model used by GNAT.
- Chapter 3 [Compiling Using gcc], page 21, describes how to compile Ada programs with `gcc`, the Ada compiler.
- Chapter 4 [Binding Using gnatbind], page 35, describes how to perform binding of Ada programs with `gnatbind`, the GNAT binding utility.
- Chapter 5 [Linking Using gnatlink], page 43, describes `gnatlink`, a program that provides for linking using the GNAT run-time library to construct a program. `gnatlink` can also incorporate foreign language object units into the executable.
- Chapter 6 [The GNAT Make Program gnatmake], page 45, describes `gnatmake`, a utility that automatically determines the set of sources needed by an Ada compilation unit, and compiles them.
- Chapter 7 [Renaming Files Using gnatchop], page 51, describes `gnatchop`, a utility that allows you to preprocess a file that contains Ada source code, and split it into several other files, one for each compilation unit.
- Chapter 10 [The cross-referencing tools gnatxref and gnatfind], page 71, discusses `gnatxref` and `gnatfind`, two tools that provide an easy way to navigate through sources.
- Chapter 11 [File Name Krunching Using gnatkr], page 79, describes the `gnatkr` file name krunching utility, used to handle shortened file names on operating systems with a limit on the length of names.
- Chapter 12 [Preprocessing Using gnatprep], page 83, describes `gnatprep`, a preprocessor utility that allows a single source file to be used to generate multiple or parametrized source files, by means of macro substitution.
- Chapter 14 [The GNAT library browser gnatls], page 89, describes `gnatls`, a utility that displays information about compiled units, including dependences on the correspondings sources files, and consistency of compilations.
- Chapter 15 [Finding memory problems with gnatmem], page 93, describes `gnatmem`, a utility that monitors dynamic allocation and deallocation activity in a program, and displays information about incorrect deallocations and sources of possible memory leaks.
- Chapter 16 [ASIS-Based Tools], page 99, gives the general idea about the tools built on top of the ASIS implementation for GNAT.

- Chapter 17 [Creating Sample Bodies Using gnatstub], page 101, discusses `gnatstub`, an utility that generates empty, but compilable bodies for library units.
- Chapter 18 [Minimizing Executables for Ada Programs Using gnatelim], page 103, discusses `gnatelim`, a tool which detects unused subprograms and produces information that helps the compiler to create a smaller executable for a program.
- Chapter 19 [Other Utility Programs], page 107, discusses several other GNAT utilities, including `gnatpsta` and `gnatpsys`.
- Chapter 20 [Running and Debugging Ada Programs], page 111, describes how to run and debug Ada programs.
- Chapter 21 [Performance Considerations], page 119, reviews the trade offs between using defaults or options in program development.

What You Should Know Before Reading This Guide

This user's guide assumes that you are familiar with Ada 95 language, as described in the International Standard ANSI/ISO/IEC-8652:1995, Jan 1995.

Related Information

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

- *GNAT Reference Manual*, which contains all reference material for the GNAT implementation of Ada 95.
- *Ada 95 Language Reference Manual*, which contains all reference material for the Ada 95 programming language.
- *Debugging with GDB* contains all details on the use of the GNU source-level debugger.
- *GNU Emacs Manual* contains full information on the extensible editor and programming environment Emacs.

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`', '`button names`', and '`field names`'.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters "`$ `" (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the `$` replaced by whatever prompt character you are using.

1 Getting Started With GNAT

This chapter describes the simplest ways of using GNAT to compile Ada programs.

1.1 Running GNAT

Three steps are needed to create an executable file from an Ada source file:

1. The source file(s) must be compiled.
2. The file(s) must be bound using the GNAT binder.
3. All appropriate object files must be linked to produce an executable.

All three steps are most commonly handled by using the `gnatmake` utility program that, given the name of the main program, automatically performs the necessary compilation, binding and linking steps.

1.2 Running a Simple Ada Program

Any editor may be used to prepare an Ada program. If `emacs` is used, the optional Ada mode may be helpful in laying out the program. The program text is a normal text file. We will suppose in our initial example that you have used your editor to prepare the following standard format text file:

```
with Text_IO; use Text_IO;
procedure Hello is
begin
   Put_Line ("Hello WORLD!");
end Hello;
```

This file should be named '`hello.adb`'. Using the normal default file naming conventions, By default, GNAT requires that each file contain a single compilation unit whose file name corresponds to the unit name with periods replaced by hyphens, and whose extension is '`.ads`' for a spec and '`.adb`' for a body. This default file naming convention can be overridden by use of the special pragma `Source_File_Name` see Section 2.4 [Using Other File Names], page 11. 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 compilation requirements, then the `gnatchop` utility can be conveniently used to perform this renaming operation see Chapter 7 [Renaming Files Using gnatchop], page 51).

You can compile the program using the following command:

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

`gcc` is the command used to access the compiler. This compiler is capable of compiling programs in several languages including Ada 95 and C. It determines you have given it an Ada program by the extension ('`.ads`' or '`.adb`'), and will call the GNAT compiler to compile the specified file.

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

This compile command generates a file '`hello.o`' which is the object file corresponding to your Ada program. It also generates a file '`hello.ali`' which contains additional information used to check that an Ada program is consistent. To get an executable file, we then use `gnatbind` to bind the program and `gnatlink` to link it to produce the executable.

```
$ gnatbind hello.ali
$ gnatlink hello.ali
```

A simpler method of carrying out these steps is to use `gnatmake`, which is a master program which invokes all of the required compilation, binding and linking tools in the correct order. In particular, `gnatmake` automatically recompiles any sources that have been modified since they were last compiled, or sources that depend on such modified sources, so that a consistent compilation is ensured.

```
$ gnatmake hello.adb
```

The result is an executable program called '`hello`', which can be run by entering:

```
./hello
```

and, if all has gone well, you will see

```
Hello WORLD!
```

appear in response to this command.

1.3 Running a Program With Multiple Units

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

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

with Text_IO; use 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

To build an executable version of this program, we could use four separate steps to compile, bind, and link the program, as follows:

```
$ gcc -c gmain.adb
$ gcc -c greetings.adb
$ gnatbind gmain.ali
$ gnatlink gmain.ali
```

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 a separate body, only the body need be compiled. If you want to submit these programs to the compiler for semantic checking purposes, then you use the `-gnatc` switch:

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

Although the compilation can be done in separate steps as in the above example, in practice it is almost always more convenient to use the `gnatmake` capability. All you need to know in this case is the name of the main program source file. The effect of the above four commands can be achieved with a single one:

```
$ gnatmake gmain.adb
```

In the next section we discuss the advantages of using `gnatmake` in more detail.

1.4 Using the gnatmake Utility

If you work on a program by compiling single components at a time using `gcc`, you typically keep track of the units you modify. In order to build a consistent system, you compile not only these units, but also any units that depend on the units you have modified. For example, in the preceding case, if you edit '`gmain.adb`', you only need to recompile that file. But if you edit '`greetings.ads`', you must recompile both '`greetings.adb`' and '`gmain.adb`', because both files contain units that depend on '`greetings.ads`'.

`gnatbind` will warn you if you forget one of these compilation steps, so that it is impossible to generate an inconsistent program as a result of forgetting to do a compilation. Nevertheless it is tedious and error-prone to keep track of dependencies among units. One approach to handle the dependency-bookkeeping is to use a make file. However, make files present maintenance problems of their own: if the dependencies change as you change the program, you must make sure that the make file is kept up-to-date manually, which is also an error-prone process.

The `gnatmake` utility takes care of these details automatically. Invoke it using either one of the following forms:

```
$ gnatmake gmain.adb
$ gnatmake gmain
```

The argument is the name of the file containing the main program from which you may omit the extension. `gnatmake` examines the environment, automatically recompiles any files that need recompiling, and binds and links the resulting set of object files, generating the executable file, '`gmain`'. In a large program, it can be extremely helpful to use `gnatmake`, because working out by hand what needs to be recompiled can be difficult.

Note that `gnatmake` takes into account all the intricate Ada 95 rules that establish dependencies among units. These include dependencies that result from inlining subprogram bodies, and from generic instantiation. Unlike some other Ada make tools, `gnatmake` does not rely on the dependencies that were found by the compiler on a previous compilation, which may possibly be wrong when sources change. `gnatmake` determines the exact set of dependencies from scratch each time it is run.

2 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 library. The model is initially described without reference to the library-based model. If you have not previously used an Ada compiler, you need only read the first part of this chapter. The last section describes and discusses the differences between the GNAT model and the traditional Ada compiler models. If you have used other Ada compilers, this section will help you to understand those differences, and the advantages of the GNAT model.

2.1 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 Section 2.2 [Foreign Language Representation], page 7 for support of non-USA character sets). The format effector characters are represented using their standard ASCII encodings, as follows:

`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 will recognize a wide variety of stream formats, in which the end of physical physical lines is marked by any of the following sequences: `LF`, `CR`, `CR-LF`, or `LF-CR`. This is useful in accomodating files that are imported from other operating systems.

The end of a source file is normally represented by the physical end of file. However the control character `16#1A#` (`SUB`) is also recognized as signalling the end of the source file. Again, this is provided for compatibility with other operating systems where this code is used to represent the end of file.

Each file contains a single Ada compilation unit, including any pragmas associated with the unit. For example, this means you must place a package declaration (a package *spec*) and the corresponding body in separate files. An Ada *compilation* (which is a sequence of compilation units) is represented using a sequence of files. Similarly, you will place each subunit or child unit in a separate file.

2.2 Foreign Language Representation

GNAT supports the standard character sets defined in Ada 95 as well as several other non-standard character sets for use in localized versions of the compiler.

2.2.1 Latin-1

The basic character set is Latin-1. This character set is defined by ISO standard 8859, part 1. The lower half (character codes `16#00#` ... `16#7F#)` is identical to standard ASCII coding, but the upper half is used to represent additional characters. These include extended letters used by European languages, such as French accents, the vowels with umlauts used in German, and the extra letter A-ring used in Swedish.

For a complete list of Latin-1 codes and their encodings, see the source file of library unit `Ada.Characters.Latin_1` in file '`a-chlat1.ads`'. You may use any of these extended characters freely in character or string literals. In addition, the extended characters that represent letters can be used in identifiers.

2.2.2 Other 8-Bit Codes

GNAT also supports several other 8-bit coding schemes:

Latin-2 Latin-2 letters allowed in identifiers, with uppercase and lowercase equivalence.

Latin-3 Latin-3 letters allowed in identifiers, with uppercase and lowercase equivalence.

Latin-4 Latin-4 letters allowed in identifiers, with uppercase and lowercase equivalence.

IBM PC (code page 437)
This code page is the normal default for PCs in the U.S. It corresponds to the original IBM PC character set. This set has some, but not all, of the extended 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 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.

2.2.3 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. Only characters defined in the JIS code set table can be used 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. Only characters defined in the JIS code set table can be used 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 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, but in this implementation, all UTF-8 sequences of four or more bytes length will be treated as illegal).

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#`. It is also possible (though not required) to use the Brackets coding for upper half characters. For example, the code `16#A3#` can be represented 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 the standard ACVC (Ada Compiler Validation Capability) test suite distributions.

Note: Some of these coding schemes do not permit the full use of the Ada 95 character set. For example, neither Shift JIS, nor EUC allow the use of the upper half of the Latin-1 set.

2.3 File Naming Rules

The default 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.

An exception arises 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 minus. In this case, the character tilde is used in place of the minus. The reason for this special rule is to avoid 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 list shows some examples of these rules.

'`main.ads`'
Main (spec)

'`main.adb`'
Main (body)

'`arith_functions.ads`'
Arith_Functions (package spec)

'`arith_functions.adb`'
Arith_Functions (package body)

'`func-spec.ads`'
Func.Spec (child package spec)

'`func-spec.adb`'
Func.Spec (child package body)

'`main-sub.adb`'
Sub (subunit of Main)

'a~bad.adb'
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"). This may be particularly useful when programs being developed with GNAT are to be used on operating systems with limited file name lengths. See Section 11.2 [Using gnatkr], page 79.

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. See (see Chapter 7 [Renaming Files Using gnatchop], page 51) for details.

2.4 Using Other File Names

In the previous section, we have described the default rules used by GNAT to determine the file name in which a given unit resides. It is often 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 the programmer 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_Utilities.Stacks,
  Spec_File_Name => "myutilst_a.ada");
pragma Source_File_name (My_Utilities.Stacks,
  Body_File_Name => "myutilst.ada");
```

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 it will be placed 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 See Chapter 8 [Handling of Configuration Pragmas], page 55.

GNAT allows completely arbitrary file names to be specified using the source file name pragma. However, if the file name specified has an extension other than '.ads' '.adb' or '.ada' it is necesary to use a special syntax when compiling the file. The name in this case must be preceded by the special sequence -x followed by a space, as in:

```
$ gcc -c -x 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, then it must be included in the gnatmake command, it may not be omitted.

2.5 Generating Object Files

An Ada program consists of a set of source files, and the first step in compiling the program is to generate the corresponding object files. These are generated 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. The source file for the package spec need not be compiled in this case because there is 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. The spec, if one is present, is as usual in a separate file, and need not be compiled.
- In the case of subunits, only compile the parent unit. A single object file is generated 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 instantation by means of macro expansion. However, it is still necessary to compile generic units, for dependency checking and elaboration purposes.

The preceding rules describe the set of files that must be compiled to generate the object files for a program. 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.

2.6 Source Dependencies

A given object file clearly depends on 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 parents of `with`'ed child units or they are run-time units required by the language constructs used in a particular unit.
- If a file being compiled instantiates a library level generic unit, the object file depends on both the spec and body files for this generic unit.
- If a file being compiled instantiates a generic unit defined within a package, the object file depends on the body file for the package as well as the spec file.
- If a file being compiled contains a call to a subprogram for which pragma `Inline` applies and inlining is activated with the `-gnatn` switch, the object file depends on the file containing the body of this subprogram as well as on the file containing the spec. Similarly if the `-gnatN` switch is used, then the unit is dependent on all body files.
- The object file for a parent unit depends on all its subunit body files.

These rules are applied transitively: if unit `A` `with`'s unit `B`, whose elaboration calls an inlined procedure in package `C`, the object file for unit `A` will depend on the body of `C`, in file '`c.adb`'.

The set of dependent files described by these rules includes all the files on which the unit is semantically dependent, as described in the Ada 95 Language Reference Manual. However it is a superset of what the ARM describes, because it includes generic, inline, and subunit dependencies.

An object file must be recreated by recompiling the corresponding source file if any of the source files on which it depends are modified. For example, if the `make` utility is used to control compilation, the rule for an Ada object file must mention all the source files on which the object file depends, according to the above definition. The determination of the necessary recompilations is done automatically when one uses `gnatmake`.

2.7 The Ada Library Information Files

Each compilation actually generates two output files. The first of these is the normal object file that has a '`.o`' extension. The second is a text file containing full dependency information. It has the same name as the source file, but an '`.ali`' extension. This file is known as the Ada Library Information (ALI) file.

Normally you need not be concerned with the contents of this file. This section is included in case you want to understand how these files are being used by the binder and other GNAT utilities. Each ALI file consists of a series of lines of the form:

Key_Character parameter parameter ...

The first two lines in the file identify the library output version and `Standard` version. These are required to be consistent across the entire set of compilation units in your program.

```
V "xxxxxxxxxxxxxxxx"
```

This line indicates the library output version, as defined in '`gnatvsn.ads`'. It ensures that separate object modules of a program are consistent. The library output version must be changed if anything in the compiler changes that would affect successful binding of modules compiled separately. Examples of such changes are modifications in the format of the library information described in this package, modifications to calling sequences, or to the way data is represented.

```
S "xxxxxxxxxxxxxxxx"
```

This line contains information regarding types declared in packages `Standard` as stored in `Gnatvsn.Standard_Version`. The purpose of this information is to ensure that all units in a program are compiled with a consistent set of options. This is critical on systems where, for example, the size of `Integer` can be set by command line switches.

```
M type [priority]
```

This line is present only for a unit that can be a main program. *type* is either `P` for a parameterless procedure or `F` for a function returning a value of integral type. The latter is for writing a main program that returns an exit status. *priority* is present only if there was a valid pragma `Priority` in the corresponding unit to set the main task priority. It is an unsigned decimal integer.

```
F x
```

This line is present if a pragma Float_Representation or Long_Float is used to specify other than the default floating-point format. This option applies only to implementations of GNAT for the Digital Alpha Systems. The character x is 'I' for IEEE_Float, 'G' for VAX_Float with Long_Float using G_Float, and 'D' for VAX_Float for Long_Float with D_Float.

```
P L=x Q=x T=x
```

This line is present if the unit uses tasking directly or indirectly, and has one or more valid xxx_Policy pragmas that apply to the unit. The arguments are as follows

```
L=x (locking policy)
```

This is present if a valid Locking_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. '`C`' for Ceiling_Locking).

```
Q=x (queuing policy)
```

This is present if a valid Queuing_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. '`P`' for Priority_Queuing).

```
T=x (task_dispatching policy)
```

This is present if a valid Task_Dispatching_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. '`F`' for FIFO_Within_Priorities).

Following these header lines is a set of information lines, one per compilation unit. Each line lists a unit in the object file corresponding to this ALI file. In particular, when a package body or subprogram body is compiled there will be two such lines, one for the spec and one for the body, with the entry for the body appearing first. This is the only case in which a single ALI file contains more than one unit. Note that subunits do not count as compilation units for this purpose, and generate no library information, because they are inlined. The lines for each compilation unit have the following form:

```
U unit-name source-name version [attributes]
```

This line identifies the unit to which this section of the library information file applies. *unit-name* is the unit name in internal format, as described in package `Uname`, and *source-file* is the name of the source file containing the unit.

version is the version, given by eight hexadecimal characters with lowercase letters. This value is a hash code that includes contributions from the time stamps of this unit and all the units on which it semantically depends.

The optional *attributes* are a series of two-letter codes indicating information about the unit. They indicate the nature of the unit and they summarize information provided by categorization pragmas.

`EB` Unit has pragma Elaborate_Body.

`NE` Unit has no elaboration routine. All subprogram specs are in this category, as are subprogram bodies if access-before-elaboration checks are being generated. Package bodies and specs may or may not have `NE` set, depending on whether or not elaboration code is required.

`PK` Unit is a package.

`PU` Unit has pragma `Pure`.

`PR` Unit has pragma `Preelaborate`.

`RC` Unit has pragma `Remote_Call_Interface`.

`RT` Unit has pragma `Remote_Types`.

`SP` Unit has pragma `Shared_Passive`.

`SU` Unit is a subprogram.

The attributes may appear in any order, separated by spaces. The next set of lines in the ALI file have the following form:

```
W unit-name [source-name lib-name [E] [EA] [ED]]
```

One of these lines is present for each unit mentioned in an explicit `with` clause in the current unit. *unit-name* is the unit name in internal format. *source-name* is the file name of the file that must be compiled to compile that unit (usually the file for the body, except for packages that have no body). *lib-name* is the file name of the library information file that contains the results of compiling the unit. The `E` and `EA` parameters are present if pragma `Elaborate` or pragma `Elaborate_All`, respectively, apply to this unit. `ED` is used to indicate that the compiler has determined that a pragma `Elaborate_All` for this unit would be desirable. For details on the use of the ED parameter see See Chapter 9 [Elaboration Order Handling in GNAT], page 57.

Following the unit information is an optional series of lines that indicate the usage of pragma `Linker_Options`. For each appearance of pragma `Linker_Options` in any of the units for which unit lines are present, a line of the form

```
L string
```

appears. *string* is the string from the pragma enclosed in quotes. Within the quotes, the following can occur:

- 7-bit graphic characters other than `"` or `{`
- `""` (indicating a single `"` character)
- `{hh}` indicating a character whose code is hex hh

For further details, see `Stringt.Write_String_Table_Entry` in the file '`stringt.ads`'. Note that wide characters of the form `{hhhh}` cannot be produced, because `pragma Linker_Option` accepts only `String`, not `Wide_String`.

Finally, the rest of the ALI file contains a series of lines that indicate the source files on which the compiled units depend. This is used by the binder for consistency checking and looks like:

`D` *source-name time-stamp* `[`*comments*`]`

where *comments*, if present, must be separated from the time stamp by at least one blank. Currently this field is unused.

Blank lines are ignored when the library information is read, and separate sections of the file are separated by blank lines to help readability. Extra blanks between fields are also ignored.

2.8 Representation of Time Stamps

All compiled units are marked with a time stamp, which is derived from the source file. The binder uses these time stamps to ensure consistency of the set of units that constitutes a single program. Time stamps are fourteen-character strings of the form *YYYYMMDDHHMMSS*. The fields have the following meaning:

`YYYY`	year (4 digits)
`MM`	month (2 digits 01-12)
`DD`	day (2 digits 01-31)
`HH`	hour (2 digits 00-23)
`MM`	minutes (2 digits 00-59)
`SS`	seconds (2 digits 00-59)

Time stamps may be compared lexicographically (in other words, the order of Ada comparison operations on strings) to determine which is later or earlier. However, in normal mode, only equality comparisons have any effect on the semantics of the library. Later/earlier comparisons are used only for determining the most informative error messages to be issued by the binder.

The time stamp is the actual stamp stored with the file without any adjustment resulting from time zone comparisons. This avoids problems in using libraries across networks with clients spread across multiple time zones, but it means that the time stamp might differ from that displayed in a directory listing. For example, in UNIX systems, file time stamps are stored in Greenwich Mean Time (GMT), but the `ls` command displays local times.

2.9 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 it is possible to 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 it 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 C, that contains calls to the elaboration procedures of those compilation unit that require them, followed by a call to the main program. This C program is compiled using the C compiler to generate the object file for the main program. The name of the C file is `b_xxx.c` where *xxx* is the name of the main program unit. (In future versions of GNAT, this main program may be created directly in Ada).

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.

2.10 Mixed Language Programming

There are two ways to build a program that contains some Ada files and some other language files (depending on whether the main program is in Ada or not). If the main program is in Ada, one proceeds as follows:

1. Compile the Ada units to produce a set of object files and ALI files.
2. Compile the other language files to generate object files.
3. Run the Ada binder on the Ada main program.
4. Compile the Ada main program.
5. Link the Ada main program, the Ada objects and the other language objects.

If the main program is in some language other than Ada, you must use a special option of the binder to generate callable routines to initialize and finalize the Ada units (see Section 4.6 [Binding for Non-Ada Main Programs], page 39). Calls to the initialization and finalization routines must be inserted in the main program, or some other appropriate point in the code. The call to initialize the Ada units must occur before the first Ada subprogram is called, and the call to finalize the Ada units must occur after the last Ada subprogram returns. You use the same procedure for building the program as described previously. In this case, however, the binder places the initialization and finalization subprograms into file '`b_xxx.c`' instead of the main program.

GNAT follows standard calling sequence conventions and will thus interface to any other language that also follows these conventions. The following Convention identifiers are recognized by GNAT:

- Ada. This indicates that the standard Ada calling sequence will be used and all Ada data items may be passed without any limitations in the case where GNAT is used to generate both the caller and callee. It is also possible to mix GNAT generated code and code generated by another Ada compiler. In this case, the data types should be restricted to simple cases, including primitive types. Whether complex data types can be passed depends on the situation. Probably it is safe to pass simple arrays, such as arrays of integers or floats. Records may or may not work, depending on whether both compilers lay them out identically. Complex structures involving variant records, access parameters, tasks, or protected types, are unlikely to be able to be passed.

Note that in the case of GNAT running on a platform that supports DEC Ada 83, a higher degree of compatibility can be guaranteed, and in particular records are layed out in an identical manner in the two compilers. Note also that if output from two different compilers is mixed, the program is responsible for dealing with elaboration issues. Probably the safest approach is to write the main program in the version of Ada other than GNAT, so that it takes care of its own elaboration requirements, and then call the GNAT-generated adainit procedure to ensure elaboration of the GNAT components. Consult the documentation of the other Ada compiler for further details on elaboration.

However, it is not possible to mix the tasking runtime of GNAT and DEC Ada 83, All the tasking operations must either be entirely within GNAT compiled sections of the program, or entirely within DEC Ada 83 compiled sections of the program.

- Asm. Equivalent to Ada
- Assembler. Equivalent to Ada
- COBOL. Data will be passed according to the conventions described in section B.4 of the Ada 95 Reference Manual.
- C. Data will be passed according to the conventions described in section B.3 of the Ada 95 Reference Manual.
- CPP. This stands for C++. For most purposes this is identical to C. See separate description of the specialized GNAT pragmas relating to C++ intefacing for further details.
- Fortran. Data will be passed according to the conventions described in section B.5 of the Ada 95 Reference Manual.
- Intrinsic. This defines an intrinsic operation, as defined in the Ada 95 Reference Manual. Normally this is not used in application programs. The one exception is that GNAT permits the use of Intrinsic for defining shift operations on user defined (signed and unsigned) integer types.
- Stdcall. This is relevant only to NT/Win95 implementations of GNAT, and specifies that the Stdcall calling sequence will be used, as defined by the NT API.
- Stubbed. This is a special convention that indicates that the compiler should provide a stub body that raises Program_Error.

2.11 Comparison between GNAT and C/C++ Compilation Models

The GNAT model of compilation is close to the C and C++ models. You can think of Ada specs as corresponding to header files in C. As in C, you don't need to compile specs; they are compiled when they are used. The Ada `with` is similar in effect to the `#include` of a C header.

One notable difference is that, in Ada, you may compile specs separately to check them for semantic and syntactic accuracy. This is not always possible with C headers because they are fragments of programs that have less specific syntactic or semantic rules.

The other major difference is the requirement for running the binder, which performs two important functions. First, it checks for consistency. In C or C++, the only defense against assembling inconsistent programs lies outside the compiler, in a make file, for example. The binder satisfies the Ada requirement that it be impossible to construct an inconsistent program when the compiler is used in normal mode.

The other important function of the binder is to deal with elaboration issues. There are also elaboration issues in C++ that are handled automatically. This automatic handling has the advantage of being simpler to use, but the C++ programmer has no control over elaboration. Where `gnatbind` might complain there was no valid order of elaboration, a C++ compiler would simply construct a program that malfunctioned at run time.

2.12 Comparison between GNAT and Conventional Ada Library Models

This section is intended to be useful to Ada programmers who have previously used an Ada compiler implementing the traditional Ada library model, as described in the Ada 95 Language Reference Manual. If you have not used such a system, please go on to the next section.

In GNAT, there is no *library* in the normal sense. Instead, the set of source files themselves acts as the library. Compiling Ada programs does not generate any centralized information, but rather an object file and a ALI file, which are of interest only to the binder and linker. In a traditional system, the compiler reads information not only from the source file being compiled, but also from the centralized library. This means that the effect of a compilation depends on what has been previously compiled. In particular:

- When a unit is `with`'ed, the unit seen by the compiler corresponds to the version of the unit most recently compiled into the library.
- Inlining is effective only if the necessary body has already been compiled into the library.
- Compiling a unit may obsolete other units in the library.

In GNAT, compiling one unit never affects the compilation of any other units because the compiler reads only source files. Only changes to source files can affect the results of a compilation. In particular:

- When a unit is `with`'ed, the unit seen by the compiler corresponds to the source version of the unit that is currently accessible to the compiler.
- Inlining requires the appropriate source files for the package or subprogram bodies to be available to the compiler. Inlining is always effective, independent of the order in which units are complied.

- Compiling a unit never affects any other compilations. The editing of sources may cause previous compilations to be out of date if they depended on the source file being modified.

The most important result of these differences is that order of compilation is never significant in GNAT. There is no situation in which one is required to do one compilation before another. What shows up as order of compilation requirements in the traditional Ada library becomes, in GNAT, simple source dependencies; in other words, there is only a set of rules saying what source files must be present when a file is compiled.

3 Compiling Using gcc

This chapter discusses how to compile Ada programs using the gcc command. It also describes the set of switches that can be used to control the behavior of the compiler.

3.1 Compiling Programs

The first step in creating an executable program is to compile the units of the program using the gcc command. You must compile the following files:

- the body file ('.adb') for a library level subprogram or generic subprogram
- the spec file ('.ads') for a library level package or generic package that has no body
- the body file ('.adb') for a library level package or generic package that has a body

You need *not* compile the following files

- the spec of a library unit which has a body
- subunits

because they are compiled as part of compiling related units. GNAT package specs when the corresponding body is compiled, and subunits when the parent is compiled. If you attempt to compile any of these files, you will get one of the following error messages (where fff is the name of the file you compiled):

```
No code generated for file fff (package spec)
No code generated for file fff (subunit)
```

The basic command for compiling a file containing an Ada unit is

```
$ gcc -c [switches] 'file name'
```

where *file name* is the name of the Ada file (usually having an extension '.ads' for a spec or '.adb' for a body). You specify the -c switch to tell gcc to compile, but not link, the file. The result of a successful compilation is an object file, which has the same name as the source file but an extension of '.o' and an Ada Library Information (ALI) file, which also has the same name as the source file, but with '.ali' as the extension. GNAT creates these two output files in the current directory, but you may specify a source file in any directory using an absolute or relative path specification containing the directory information.

gcc is actually a driver program that looks at the extensions of the file arguments and loads the appropriate compiler. For example, the GNU C compiler is 'cc1', and the Ada compiler is 'gnat1'. These programs are in directories known to the driver program (in some configurations via environment variables you set), but need not be in your path. The gcc driver also calls the assembler and any other utilities needed to complete the generation of the required object files.

It is possible to supply several file names on the same gcc command. This causes gcc to call the appropriate compiler for each file. For example, the following command lists three separate files to be compiled:

```
$ gcc -c x.adb y.adb z.c
```

calls gnat1 (the Ada compiler) twice to compile 'x.adb' and 'y.adb', and cc1 (the C compiler) once to compile 'z.c'. The compiler generates three object files 'x.o', 'y.o' and '`z.o`' and the two ALI files '`x.ali`' and '`y.ali`' from the Ada compilations. Any switches apply to all the files listed, except for `-gnat`*x* switches, which apply only to Ada compilations.

3.2 Switches for `gcc`

The `gcc` command accepts numerous switches to control the compilation process. These switches are fully described in this section.

`-b` *target*
: Compile your program to run on *target*, which is the name of a system configuration. You must have a GNAT cross-compiler built if *target* is not the same as your host system.

`-B`*dir*
: Load compiler executables (for example, `gnat1`, the Ada compiler) from *dir* instead of the default location. Only use this switch when multiple versions of the GNAT compiler are available. See the `gcc` manual page for further details. You would normally use the `-b` or `-V` switch instead.

`-c`
: Compile. Always use this switch when compiling Ada programs.

Note that you may not use `gcc` without a `-c` switch to compile and link in one step. This is because the binder must be run, and currently `gcc` cannot be used to run the GNAT binder.

`-g`
: Generate debugging information. This information is stored in the object file and copied from there to the final executable file by the linker, where it can be read by the debugger. You must use the `-g` switch if you plan on using the debugger.

`-I`*dir*
: Direct GNAT to search the *dir* directory for source files needed by the current compilation (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33).

`-I-`
: Do not look for source files in the directory containing the source file named in the command line (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33).

`-o` *file*
: This switch is used in `gcc` to redirect the generated object file and its associated ALI file. Beware of this switch with GNAT, because it may cause the object file and ALI file to have different names which in turn may confuse the binder and the linker.

`-O[`*n*`]`
: *n* controls the optimization level.

n = 0
: No optimization, the default setting if no `-O` appears

n = 1
: Normal optimization, the default if you specify `-O` without an operand.

n = 2
: Extensive optimization

n = 3
: Extensive optimization with automatic inlining. This applies only to inlining within a unit. For details on control of inter-unit inlining see See Section 3.2.10 [Subprogram Inlining Control], page 31.

`-S` Used in place of `-c` to cause the assembler source file to be generated, using '`.s`' as the extension, instead of the object file. This may be useful if you need to examine the generated assembly code.

`-v` Show commands generated by the `gcc` driver. Normally used only for debugging purposes or if you need to be sure what version of the compiler you are executing.

`-V` *ver* Execute *ver* version of the compiler. This is the `gcc` version, not the GNAT version.

`-Wuninitialized`
Generate warnings for uninitialized variables. You must also specify the `-O` switch (in other words, This switch works only if optimization is turned on).

`-gnata` Assertions enabled. `Pragma Assert` and `pragma Debug` to be activated.

`-gnatb` Generate brief messages to `stderr` even if verbose mode set.

`-gnatc` Check syntax and semantics only (no code generation attempted).

`-gnate` Error messages generated immediately, not saved up till end.

`-gnatE` Full dynamic elaboration checks.

`-gnatf` Full errors. Multiple errors per line, all undefined references.

`-gnatg` GNAT style checks enabled.

`-gnati`*c* Identifier char set (*c*=1/2/3/4/8/p/f/n/w).

`-gnath` Output usage information. The output is written to `stdout`.

`-gnatk`*n* Limit file names to *n* (1-999) characters (`k` = krunch).

`-gnatl` Output full source listing with embedded error messages.

`-gnatm`*n* Limit number of detected errors to *n* (1-999).

`-gnatn` Activate inlining across unit boundaries for subprograms for which pragma `inline` is specified.

`-gnatN` Activate inlining across unit boundaries for all subprograms (not just those for which pragma `inline` is specified. This is equivalent to using `-gnatn` and adding a pragma `inline` for every subprogram in the program.

`-fno-inline`
Suppresses all inlining, even if other optimization or inlining switches are set.

`-gnato` Enable other checks, not normally enabled by default, including numeric overflow checking, and access before elaboration checks.

`-gnatp` Suppress all checks.

`-gnatq` Don't quit; try semantics, even if parse errors.

`-gnatr` Reference manual column layout required.

`-gnats` Syntax check only.

`-gnatt` Tree output file to be generated.

`-gnatu` List units for this compilation.

`-gnatv` Verbose mode. Full error output with source lines to `stdout`.

`-gnatw`*m* Warning mode (*m*=`s,e,l` for suppress, treat as error, elaboration warnings).

`-gnatW`*e* Wide character encoding method (*e*=n/h/u/s/e/8).

`-gnatz`*m* Distribution stub generation and compilation (*m*=r/c for receiver/caller stubs).

`-gnat83` Enforce Ada 83 restrictions.

`-gnat95` Standard Ada 95 mode

You may combine a sequence of GNAT switches into a single switch. For example, the combined switch

```
-gnatcfi3
```

is equivalent to specifying the following sequence of switches:

```
-gnatc -gnatf -gnati3
```

3.2.1 Error Message Control

The standard default format for error messages is called "brief format." Brief format messages are written to `stdout` (the standard output file) and have the following form:

```
e.adb:3:04: Incorrect spelling of keyword "function"
e.adb:4:20: ";" should be "is"
```

The first integer after the file name is the line number in the file, and the second integer is the column number within the line. `emacs` can parse the error messages and point to the referenced character. The following switches provide control over the error message format:

`-gnatv` The v stands for verbose. The effect of this setting is to write long-format error messages to `stdout`. The same program compiled with the `-gnatv` switch would generate:

```
3. funcion X (Q : Integer)
   |
>>> Incorrect spelling of keyword "function"
4. return Integer;
                 |
>>> ";" should be "is"
```

The vertical bar indicates the location of the error, and the '`>>>`' prefix can be used to search for error messages. When this switch is used the only source lines output are those with errors.

`-gnatl` The `l` stands for list. This switch causes a full listing of the file to be generated. The output might look as follows:

```
1. procedure E is
2.    V : Integer;
3.    function X (Q : Integer)
      |
    >>> Incorrect spelling of keyword "function"
 4.     return Integer;
                      |
    >>> ";" should be "is"
 5.    begin
 6.       return Q + Q;
 7.    end;
 8. begin
 9.    V := X + X;
10.end E;
```

When you specify the `-gnatv` or `-gnatl` switches and standard output is redirected, a brief summary is written to `stderr` (standard error) giving the number of error messages and warning messages generated.

`-gnatb`
: The `b` stands for brief. This switch causes GNAT to generate the brief format error messages to `stdout` as well as the verbose format message or full listing.

`-gnatm`*n*
: The `m` stands for maximum. *n* is a decimal integer in the range of 1 to 999 and limits the number of error messages to be generated. For example, using `-gnatm2` might yield

```
e.adb:3:04: Incorrect spelling of keyword "function"
e.adb:5:35: missing ".."
fatal error: maximum errors reached
compilation abandoned
```

`-gnatf`
: The `f` stands for full. Normally, the compiler suppresses error messages that are likely to be redundant. This switch causes all error messages to be generated. In particular, in the case of references to undefined variables. If a given variable is referenced several times, the normal format of messages is

```
e.adb:7:07: "V" is undefined (more references follow)
```

where the parenthetical comment warns that there are additional references to the variable `V`. Compiling the same program with the `-gnatf` switch yields

```
e.adb:7:07: "V" is undefined
e.adb:8:07: "V" is undefined
e.adb:8:12: "V" is undefined
e.adb:8:16: "V" is undefined
e.adb:9:07: "V" is undefined
e.adb:9:12: "V" is undefined
```

`-gnatq`
: The `q` stands for quit (really "don't quit"). In normal operation mode, the compiler first parses the program and determines if there are any syntax errors. If there are, appropriate error messages are generated and compilation is immediately terminated. This switch tells GNAT to continue with semantic analysis even if syntax errors have been found. This may enable the detection of more errors in a single run. On the other hand, the semantic analyzer is more likely to encounter some internal fatal error when given a syntactically invalid tree.

`-gnate`
: Normally, the compiler saves up error messages and generates them at the end of compilation in proper sequence. This switch (the '`e`' stands for error) causes

error messages to be generated as soon as they are detected. The use of `-gnate` usually causes error messages to be generated out of sequence. Use this switch when the compiler terminates abnormally because of an internal error. In this case, the error messages may be lost. Sometimes blowups are the result of mishandled error messages, so you may want to run with the `-gnate` switch to determine whether any error messages were generated before the crash (see Section 20.9 [GNAT Abnormal Termination], page 116).

In addition to error messages, which correspond to illegalities as defined in the Ada 95 Reference Manual, the compiler detects two kinds of warning situations.

First, the compiler considers some constructs suspicious and generates a warning message to alert you to a possible error. Second, if the compiler detects a situation that is sure to raise an exception at run time, it generates a warning message. The following shows an example of warning messages:

```
e.adb:4:24: warning: creation of object may raise Storage_Error
e.adb:10:17: warning: static value out of range
e.adb:10:17: warning: "Constraint_Error" will be raised at run time
```

GNAT considers a large number of situations as appropriate for the generation of warning messages. As always, warnings are not definite indications of errors. For example, if you do an out-of-range assignment with the deliberate intention of raising a `Constraint_Error` exception, then the warning that may be issued does not indicate an error. Some of the situations for which GNAT issues warnings (at least some of the time) are:

- Possible infinitely recursive calls
- Out-of-range values being assigned
- Possible order of elaboration problems
- Unreachable code
- Variables that are never assigned a value
- Variables that are referenced before being initialized
- Task entries with no corresponding Accept statement
- Duplicate Accepts for the same task entry in a select
- Objects that take too much storage
- Unchecked conversion between types of differing sizes
- Missing return statements along some execution paths in a function
- Incorrect pragmas
- Incorrect external names
- Allocation from empty storage pool
- Potentially blocking operations in protected types
- Suspicious parenthesization of expressions
- Mismatching bounds in an aggregate
- Attempt to return local value by reference
- Unrecognized pragmas
- Premature instantiation of a generic body

- Attempt to pack aliased components
- Out of bounds array subscripts
- Wrong length on string assignment

Three switches are available to control the handling of warning messages:

`-gnatwe` The `w` stands for warning and the `e` stands for error. This switch causes warning messages to be treated as errors. The warning string still appears, but the warning messages are counted as errors, and prevent the generation of an object file.

`-gnatws` The '`s`' stands for suppress. This switch completely suppresses the output of all warning messages.

`-gnatwl` The '`l`' stands for elaboration. This switch causes the generation of additional warning messages relating to elaboration issues. See the separate chapter on elaboration order handling for full details of the use of this switch.

3.2.2 Debugging and Assertion Control

`-gnata` The pragmas `Assert` and `Debug` normally have no effect and are ignored. This switch, where '`a`' stands for assert, causes `Assert` and `Debug` pragmas to be activated.

The pragmas have the form:

```
pragma Assert (Boolean-expression [, static-string-expression])
pragma Debug (procedure call)
```

The `Assert` pragma causes *Boolean-expression* to be tested. If the result is `True`, the pragma has no effect (other than possible side effects from evaluating the expression). If the result is `False`, the exception `Assert_Error` declared in the package `System.Assertions` is raised (passing *static-string-expression*, if present, as the message associated with the exception). If no string expression is given the default is a string giving the file name and line number of the pragma.

The `Debug` pragma causes *procedure* to be called. Note that `pragma Debug` may appear within a declaration sequence, allowing debugging procedures to be called between declarations.

3.2.3 Run-time Checks

If you compile with the default options, GNAT will insert many run-time checks into the compiled code, including code that performs range checking against constraints, but not arithmetic overflow checking for integer operations (including division by zero) or checks for access before elaboration on subprogram calls. All other run-time checks, as required by the Ada 95 Reference Manual, are generated by default. The following `gcc` switches refine this default behavior:

`-gnatp` Suppress all run-time checks as though `pragma Suppress (all_checks)` had been present in the source. Use this switch to improve the performance of the code at the expense of safety in the presence of invalid data or program bugs.

`-gnato` Enables overflow checking for integer operations. This causes GNAT to generate slower and larger executable programs by adding code to check for both overflow and division by zero (resulting in raising `Constraint_Error` as required by Ada semantics). Note that the `-gnato` switch does not affect the code generated for any floating-point operations; it applies only to integer operations. For floating-point, GNAT has the `Machine_Overflows` attribute set to `False` and the normal mode of operation is to generate IEEE NaN and infinite values on overflow or invalid operations (such as dividing 0.0 by 0.0).

`-gnatE` Enables dynamic checks for access-before-elaboration on subprogram calls and generic instantiations. For full details of the effect and use of this switch, See Chapter 3 [Compiling Using gcc], page 21.

The setting of these switches only controls the default setting of the checks. You may modify them using either `Suppress` (to remove checks) or `Unsuppress` (to add back suppressed checks) pragmas in the program source.

3.2.4 Using `gcc` for Syntax Checking

`-gnats` The `s` stands for syntax.

Run GNAT in syntax checking only mode. For example, the command

```
$ gcc -c -gnats x.adb
```

compiles file '`x.adb`' in syntax-check-only mode. You can check a series of files in a single command, and can use wild cards to specify such a group of files. Note that you must specify the `-c` (compile only) flag in addition to the `-gnats` flag.

You may use other switches in conjunction with `-gnats`. In particular, `-gnatl` and `-gnatv` are useful to control the format of any generated error messages.

The output is simply the error messages, if any. No object file or ALI file is generated by a syntax-only compilation. Also, no units other than the one specified are accessed. For example, if a unit `X` `with`'s a unit `Y`, compiling unit `X` in syntax check only mode does not access the source file containing unit `Y`.

Normally, GNAT allows only a single unit in a source file. However, this restriction does not apply in syntax-check-only mode, and it is possible to check a file containing multiple compilation units concatenated together. This is primarily used by the `gnatchop` utility (see Chapter 7 [Renaming Files Using gnatchop], page 51).

3.2.5 Using `gcc` for Semantic Checking

`-gnatc` The `c` stands for check. Causes the compiler to operate in semantic check mode, with full checking for all illegalities specified in the Ada 95 Reference Manual, but without generation of any source code (no object or ALI file generated).

Because dependent files must be accessed, you must follow the GNAT semantic restrictions on file structuring to operate in this mode:

- The needed source files must be accessible (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33).
- Each file must contain only one compilation unit.
- The file name and unit name must match (see Section 2.3 [File Naming Rules], page 10).

The output consists of error messages as appropriate. No object file or ALI file is generated. The checking corresponds exactly to the notion of legality in the Ada 95 Reference Manual.

Any unit can be compiled in semantics-checking-only mode, including units that would not normally be compiled (subunits, and specifications where a separate body is present).

3.2.6 Compiling Ada 83 Programs

`-gnat83` Although GNAT is primarily an Ada 95 compiler, it accepts this switch to specify that an Ada 83 program is to be compiled in Ada83 mode. If you specify this switch, GNAT rejects most Ada 95 extensions and applies Ada 83 semantics where this can be done easily. It is not possible to guarantee this switch does a perfect job; for example, some subtle tests, such as are found in earlier ACVC tests (that have been removed from the ACVC suite for Ada 95), may not compile correctly. However for most purposes, using this switch should help to ensure that programs that compile correctly under the `-gnat83` switch can be ported easily to an Ada 83 compiler. This is the main use of the switch.

With few exceptions (most notably the need to use `<>` on unconstrained generic formal parameters, the use of the new Ada 95 keywords, and the use of packages with optional bodies), it is not necessary to use the `-gnat83` switch when compiling Ada 83 programs, because, with rare exceptions, Ada 95 is upwardly compatible with Ada 83. This means that a correct Ada 83 program is usually also a correct Ada 95 program.

`-gnat95` This switch specifies normal Ada 95 mode, and cancels the effect of any previously given -gnat83 switch.

3.2.7 Style Checking

`-gnatr` Normally, GNAT permits any source layout consistent with the Ada 95 reference manual requirements. This switch (‘`r`’ is for "reference manual") enforces the layout conventions suggested by the examples and syntax rules of the Ada 95 Language Reference Manual. For example, an `else` must line up with an `if` and code in the `then` and `else` parts must be indented. The compiler treats violations of the layout rules as syntax errors if you specify this switch.

`-gnatg` Enforces a set of style conventions that correspond to the style used in the GNAT source code. All compiler units are always compiled with the `-gnatg` switch specified.

You can find the full documentation for the style conventions imposed by -gnatg in the body of the package `Style` in the compiler sources (in the file '`style.adb`').

You should not normally use the `-gnatg` switch. However, you *must* use -gnatg for compiling any language-defined unit, or for adding children to any language-defined unit other than `Standard`.

3.2.8 Character Set Control

`-gnati`*c* Normally GNAT recognizes the Latin-1 character set in source program identifiers, as described in the Ada 95 Reference Manual. This switch causes GNAT to recognize alternate character sets in identifiers. *c* is a single character indicating the character set, as follows:

`1`	Latin-1 identifiers
`2`	Latin-2 letters allowed in identifiers
`3`	Latin-3 letters allowed in identifiers
`4`	Latin-4 letters allowed in identifiers
`p`	IBM PC letters (code page 437) allowed in identifiers
`8`	IBM PC letters (code page 850) allowed in identifiers
`f`	Full upper-half codes allowed in identifiers
`n`	No upper-half codes allowed in identifiers
`w`	Wide-character codes allowed in identifiers

See Section 2.2 [Foreign Language Representation], page 7, for full details on the implementation of these character sets.

`-gnatW`*e* Specify the method of encoding for wide characters. *e* is one of the following:

`h`	Hex encoding (brackets coding also recognized)
`u`	Upper half encoding (brackets encoding also recognized)
`s`	Shift/JIS encoding (brackets encoding also recognized)
`e`	EUC encoding (brackets encoding also recognized)
`8`	UTF-8 encoding (brackets encoding also recognized)
`b`	Brackets encoding only (default value)

For full details on the these encoding methods see See Section 2.2.3 [Wide Character Encodings], page 9. Note that brackets coding is always accepted, even if one of the other options is specified, so for example `-gnatW8` specifies that both brackets and `UTF-8` encodings will be recognized. The units that are with'ed directly or indirectly will be scanned using the specified representation scheme, and so if one of the non-brackets scheme is used, it must be used consistently throughout the program. However, since brackets encoding is always recognized, it may be conveniently used in standard libraries, allowing these libraries to be used with any of the available coding schemes. scheme. If no `-gnatW?` parameter is present, then the default representation is Brackets encoding only.

Note that the wide character representation that is specified (explicitly or by default) for the main program also acts as the default encoding used for Wide_Text_IO files if not specifically overridden by a WCEM form parameter.

3.2.9 File Naming Control

`-gnatk`*n* Activates file name "krunching". *n*, a decimal integer in the range 1-999, indicates the maximum allowable length of a file name (not including the '`.ads`' or '`.adb`' extension). The default is not to enable file name krunching.

For the source file naming rules, See Section 2.3 [File Naming Rules], page 10.

3.2.10 Subprogram Inlining Control

`-gnatn` The `n` here is intended to suggest the first syllable of the word "inline". GNAT recognizes and processes `Inline` pragmas. However, for the inlining to actually occur, optimization must be enabled. To enable inlining across unit boundaries, this is, inlining a call in one unit of a subprogram declared in a `with`'ed unit, you must also specify this switch. In the absence of this switch, GNAT does not attempt inlining across units and does not need to access the bodies of subprograms for which `pragma Inline` is specified if they are not in the current unit.

If you specify this switch the compiler will access these bodies, creating an extra source dependency for the resulting object file, and where possible, the call will be inlined. For further details on when inlining is possible see See Section 21.3 [Inlining of Subprograms], page 120.

`-gnatN` This switch enforces a more extreme form of inlining across unit boundaries. It causes the compiler to proceed as though the normal (pragma) inlining switch was set, and to assume that there is a pragma `Inline` for every subprogram referenced by the compiled unit.

3.2.11 Auxiliary Output Control

`-gnatt` Cause GNAT to write the internal tree for a unit to a file (with the extension '`.atb`' for a body or '`.ats`' for a spec). This is not normally required, but is used by separate analysis tools. Typically these tools do the necessary compilations automatically, so you should never have to specify this switch in normal operation.

`-gnatu` Print a list of units required by this compilation on `stdout`. The listing includes all units on which the unit being compiled depends either directly or indirectly.

3.2.12 Debugging Control

`-gnatdx` Activate internal debugging switches. *x* is a letter or digit, or string of letters or digits, which specifies the type of debugging outputs desired. Normally these are used only for internal development or system debugging purposes. You can find full documentation for these switches in the body of the `Debug` unit in the compiler source file '`debug.adb`'.

One particularly useful switch is `-gnatdg`, which produces a listing of the expanded code in Ada source form. For example, all tasking constructs are reduced to appropriate run-time library calls. The syntax of this listing is close to legal Ada with the following additions:

`new` *xxx* `[storage_pool =` *yyy*`]`
Shows the storage pool being used for an allocator.

`at end` *procedure-name*`;`
Shows the finalization (cleanup) procedure for a scope.

`(if` *expr* `then` *expr* `else` *expr*`)`
Conditional expression equivalent to the `x?y:z` construction in C.

target`^(`*source*`)`
A conversion with floating-point truncation instead of rounding.

target`?(`*source*`)`
A conversion that bypasses normal Ada semantic checking. In particular enumeration types and fixed-point types are treated simply as integers.

target`?^(`*source*`)`
Combines the above two cases.

x `#/` *y*
x `#mod` *y*
x `#*` *y*
x `#rem` *y* A division or multiplication of fixed-point values which are treated as integers without any kind of scaling.

`free` *expr* `[storage_pool =` *xxx*`]`
Shows the storage pool associated with a `free` statement.

`freeze` *typename* `[`*actions*`]`
Shows the point at which *typename* is frozen, with possible associated actions to be performed at the freeze point.

`reference` *itype*
Reference (and hence definition) to internal type *itype*.

function-name`! (`*arg*`,` *arg*`,` *arg*`)`
Intrinsic function call.

labelname `: label`
Declaration of label *labelname*.

expr `&&` *expr* `&&` *expr* `...` `&&` *expr*
A multiple concatenation (same effect as *expr* & *expr* & *expr*, but handled more efficiently).

`[constraint_error]`
Raise the `Constraint_Error` exception.

expression`'reference`
A pointer to the result of evaluating *expression*.

target-type`!(`*source-expression*`)`
An unchecked conversion of *source-expression* to *target-type*.

`[`*numerator*`/`*denominator*`]`
Used to represent internal real literals (that) have no exact representation in base 2-16 (for example, the result of compile time evaluation of the expression 1.0/27.0).

3.3 Search Paths and the Run-Time Library (RTL)

With the GNAT source-based library system, the compiler must be able to find source files for units that are needed by the unit being compiled. Search paths are used to guide this process.

The compiler compiles one source file whose name must be given explicitly on the command line. In other words, no searching is done for this file. To find all other source files that are needed (the most common being the specs of units), the compiler examines the following directories, in the following order:

1. The directory containing the source file of the main unit being compiled (the file name on the command line).
2. Each directory named by an `-I` switch given on the `gcc` command line, in the order given.
3. Each of the directories listed in the value of the `ADA_INCLUDE_PATH` environment variable. Construct this value exactly as the `PATH` environment variable: a list of directory names separated by colons.
4. The default location for the GNAT Run Time Library (RTL) source files. This is determined at the time GNAT is built and installed on your system.

Specifying the switch `-I-` inhibits the use of the directory containing the source file named in the command line. You can still have this directory on your search path, but in this case it must be explicitly requested with a `-I` switch.

The compiler outputs its object files and ALI files in the current working directory. Caution: The object file can be redirected with the `-o` switch; however, `gcc` and `gnat1` have not been coordinated on this so the ALI file will not go to the right place. Therefore, you should avoid using the `-o` switch.

The packages `Ada`, `System`, and `Interfaces` and their children make up the GNAT RTL, together with the simple `System.IO` package used in the "Hello World" example. The sources for these units are needed by the compiler and are kept together in one directory.

Not all of the bodies are needed, but all of the sources are kept together anyway. In a normal installation, you need not specify these directory names when compiling or binding. Either the environment variables or the built-in defaults cause these files to be found.

In addition to the language-defined hierarchies (System, Ada and Interfaces), the GNAT distribution provides a fourth hierarchy, consisting of child units of GNAT. This is a collection of generally useful routines. See the GNAT Reference Manual for further details.

Besides simplifying access to the RTL, a major use of search paths is in compiling sources from multiple directories. This can make development environments much more flexible.

3.4 Order of Compilation Issues

If, in our earlier example, there was a spec for the `hello` procedure, it would be contained in the file '`hello.ads`'; yet this file would not have to be explicitly compiled. This is the result of the model we chose to implement library management. Some of the consequences of this model are as follows:

- There is no point in compiling specs (except for package specs with no bodies) because these are compiled as needed by clients. If you attempt a useless compilation, you will receive an error message. It is also useless to compile subunits because they are compiled as needed by the parent.
- There are no order of compilation requirements: performing a compilation never obsoletes anything. The only way you can obsolete something and require recompilations is to modify one of the source files on which it depends.
- There is no library as such, apart from the ALI files (see Section 2.7 [The Ada Library Information Files], page 13, for information on the format of these files). For now we find it convenient to create separate ALI files, but eventually the information therein may be incorporated into the object file directly.
- When you compile a unit, the source files for the specs of all units that it `with`'s, all its subunits, and the bodies of any generics it instantiates must be available (reachable by the search-paths mechanism described above), or you will receive a fatal error message.

3.5 Examples

The following are some typical Ada compilation command line examples:

```
$ gcc -c xyz.adb
```

Compile body in file '`xyz.adb`' with all default options.

```
$ gcc -c -O2 -gnata xyz-def.adb
```

Compile the child unit package in file '`xyz-def.adb`' with extensive optimizations, and pragma `Assert`/`Debug` statements enabled.

```
$ gcc -c -gnatc abc-def.adb
```

Compile the subunit in file '`abc-def.adb`' in semantic-checking-only mode.

4 Binding Using gnatbind

This chapter describes the GNAT binder, `gnatbind`, which is used to bind compiled GNAT objects. The `gnatbind` program performs four separate functions:

1. Checks that a program is consistent, in accordance with the rules in Chapter 10 of the Ada 95 Reference Manual. In particular, error messages are generated if a program uses inconsistent versions of a given unit.
2. Checks that an acceptable order of elaboration exists for the program and issues an error message if it cannot find an order of elaboration that satisfies the rules in Chapter 10 of the Ada 95 Language Manual.
3. Generates a main program incorporating the given elaboration order. This program is a small C source file that must be subsequently compiled using the C compiler. The two most important functions of this program are to call the elaboration routines of units in an appropriate order and to call the main program.
4. Determines the set of object files required by the given main program. This information is output in the forms of comments in the generated C program, to be read by the `gnatlink` utility used to link the Ada application.

4.1 Running gnatbind

The form of the `gnatbind` command is

```
$ gnatbind [switches] mainprog.ali [switches]
```

where *mainprog*.adb is the Ada file containing the main program unit body. If no switches are specified, `gnatbind` constructs a C file whose name is '`b_`*mainprog*`.c`'. For example, if given the parameter '`hello.ali`', for a main program contained in file '`hello.adb`', the binder output file would be '`b_hello.c`'.

When doing consistency checking, the binder takes any source files it can locate into consideration. For example, if the binder determines that the given main program requires the package `Pack`, whose ALI file is '`pack.ali`' and whose corresponding source spec file is '`pack.ads`', it attempts to locate the source file '`pack.ads`' (using the same search path conventions as previously described for the `gcc` command). If it can locate this source file, it checks that the time stamps or source checksums of the source and its references to in ALI files match. In other words, any ALI files that mentions this spec must have resulted from compiling this version of the source file (or in the case where the source checksums match, a version close enough that the difference does not matter).

The effect of this consistency checking, which includes source files, is that the binder ensures that the program is consistent with the latest version of the source files that can be located at bind time. Editing a source file without compiling files that depend on the source file cause error messages to be generated by the binder.

For example, suppose you have a main program '`hello.adb`' and a package `P`, from file '`p.ads`' and you perform the following steps:

1. Enter `gcc -c hello.adb` to compile the main program.
2. Enter `gcc -c p.ads` to compile package `P`.

3. Edit file '`p.ads`'.
4. Enter `gnatbind hello.ali`.

At this point, the file '`p.ali`' contains an out-of-date time stamp because the file '`p.ads`' has been edited. The attempt at binding fails, and the binder generates the following error messages:

```
error: "hello.adb" must be recompiled ("p.ads" has been modified)
error: "p.ads" has been modified and must be recompiled
```

Now both files must be recompiled as indicated, and then the bind can succeed, generating a main program. You need not normally be concerned with the contents of this file, but it is similar to the following:

```
extern int gnat_argc;
extern char **gnat_argv;
extern char **gnat_envp;
extern int gnat_exit_status;
void adafinal ();
void adainit ()
{
   __gnat_set_globals (
      -1,    /* Main_Priority              */
      -1,    /* Time_Slice_Value           */
      ' ',   /* Locking_Policy             */
      ' ',   /* Queuing_Policy             */
      ' ',   /* Tasking_Dispatching_Policy */
      adafinal);
   system___elabs ();
/* system__standard_library___elabs (); */
/* system__task_specific_data___elabs (); */
/* system__tasking_soft_links___elabs (); */
   system__tasking_soft_links___elabb ();
/* system__task_specific_data___elabb (); */
/* system__standard_library___elabb (); */
/* m___elabb (); */
}
void adafinal () {
}
int main (argc, argv, envp)
    int argc;
    char **argv;
    char **envp;
{
   gnat_argc = argc;
   gnat_argv = argv;
   gnat_envp = envp;

   __gnat_initialize();
   adainit();
```

```
      _ada_m ();

      adafinal();
      __gnat_finalize();
      exit (gnat_exit_status);
   }
   unsigned mB = 0x2B0EB17F;
   unsigned system__standard_libraryB = 0x0122ED49;
   unsigned system__standard_libraryS = 0x79B018CE;
   unsigned systemS = 0x08FBDA7E;
   unsigned system__task_specific_dataB = 0x6CC7367B;
   unsigned system__task_specific_dataS = 0x47178527;
   unsigned system__tasking_soft_linksB = 0x5A75A73C;
   unsigned system__tasking_soft_linksS = 0x3012AFCB;
   /* BEGIN Object file/option list
   ./system.o
   ./s-tasoli.o
   ./s-taspda.o
   ./s-stalib.o
   ./m.o
      END Object file/option list */
```

The call to `__gnat_set_globals` establishes program parameters, including the priority of the main task, and parameters for tasking control. It also passes the address of the finalization routine so that it can be called at the end of program execution.

Next there is code to save the `argc` and `argv` values for later access by the `Ada.Command_Line` package. The variable `gnat_exit_status` saves the exit status set by calls to `Ada.Command_Line.Set_Exit_Status` and is used to return an exit status to the system.

The call to `__gnat_initialize` and the corresponding call at the end of execution to `__gnat_finalize` allow any specialized initialization and finalization code to be hooked in. The default versions of these routines do nothing.

The calls to `xxx___elabb` and `xxx___elabs` perform necessary elaboration of the bodies and specs respectively of units in the program. These calls are commented out if the unit in question has no elaboration code.

The call to `m` is the call to the main program.

The list of unsigned constants gives the version number information. Version numbers are computed by combining time stamps of a unit and all units on which it depends. These values are used for implementation of the `Version` and `Body_Version` attributes.

Finally, a set of comments gives the full names of all the object files that must be linked to provide the Ada component of the program. As seen in the previous example, this list includes the files explicitly supplied and referenced by the user as well as implicitly referenced run-time unit files. The latter are omitted if the corresponding units reside in shared libraries. The directory names for the run-time units depend on the system configuration.

4.2 Consistency-Checking Modes

As described in the previous section, by default `gnatbind` checks that object files are consistent with one another and are consistent with any source files it can locate. The following switches control binder access to sources.

`-s` Require source files to be present. In this mode, the binder must be able to locate all source files that are referenced, in order to check their consistency. In normal mode, if a source file cannot be located it is simply ignored. If you specify this switch, a missing source file is an error.

`-x` Exclude source files. In this mode, the binder only checks that ALI files are consistent with one another. Source files are not accessed. The binder runs faster in this mode, and there is still a guarantee that the resulting program is self-consistent. If a source file has been edited since it was last compiled, and you specify this switch, the binder will not detect that the object file is out of date with respect to the source file. Note that this is the mode that is automatically used by `gnatmake` because in this case the checking against sources has already been performed by `gnatmake` in the course of compilation (i.e. before binding).

4.3 Binder Error Message Control

The following switches provide control over the generation of error messages from the binder:

`-v` Verbose mode. In the normal mode, brief error messages are generated to `stderr`. If this switch is present, a header is written to `stdout` and any error messages are directed to `stdout`. All that is written to `stderr` is a brief summary message.

`-b` Generate brief error messages to `stderr` even if verbose mode is specified. This is relevant only when used with the `-v` switch.

`-m`*n* Limits the number of error messages to *n*, a decimal integer in the range 1-999. The binder terminates immediately if this limit is reached.

`-r` Renames the generated main program from `main` to `gnat_main`. This is useful in the case of some cross-building environments, where the actual main program is separate from the one generated by `gnatbind`.

`-ws` Suppress all warning messages.

`-we` Treat any warning messages as fatal errors.

`-t` Ignore time stamp errors. Any time stamp error messages are treated as warning messages. This switch essentially disconnects the normal consistency checking, and the resulting program may have undefined semantics if inconsistent units are present. *This means that* `-t` *should be used only in unusual situations, with extreme care.*

4.4 Elaboration Control

The following switches provide additional control over the elaboration order. For full details see See Chapter 9 [Elaboration Order Handling in GNAT], page 57.

`-f` Instructs the binder to ignore directives from the compiler about implied `Elaborate_All` pragmas, and to use full Ada 95 Reference Manual semantics in an attempt to find a legal elaboration order, even if it seems likely that this order will cause an elaboration exception.

`-p` Normally the binder attempts to choose an elaboration order that is likely to minimize the likelihood of an elaboration order error resulting in raising a `Program_Error` exception. This switch reverses the action of the binder, and requests that it deliberately choose an order that is likely to maximize the likelihood of an elaboration errror. This is useful in ensuring portability and avoiding dependence on accidental fortuitous elaboration ordering.

4.5 Output Control

The following switches allow additional control over the output generated by the binder.

`-e` Output complete list of elaboration-order dependencies, showing the reason for each dependency. This output can be rather extensive but may be useful in diagnosing problems with elaboration order. The output is written to `stdout`.

`h` Output usage information. The output is written to `stdout`.

`-l` Output chosen elaboration order. The output is written to `stdout`.

`-o` *file* Set name of output file to *file* instead of the normal '`b_`*prog*`.c`' default. You would normally give *file* an extension of '`.c`' because it will be a C source program.

`-c` Check only. Do not generate the binder output file. In this mode the binder performs all error checks but does not generate an output file.

4.6 Binding for Non-Ada Main Programs

In our description so far we have assumed that the main program is in Ada, and that the task of the binder is to generate a corresponding function `main` that invokes this Ada main program. GNAT also supports the building of executable programs where the main program is not in Ada, but some of the called routines are written in Ada and compiled using GNAT (see Section 2.10 [Mixed Language Programming], page 17). The following switch is used in this situation:

`-n` No main program. The main program is not in Ada.

In this case, most of the functions of the binder are still required, but instead of generating a main program, the binder generates a file containing the following callable routines:

`adainit` You must call this routine to initialize the Ada part of the program by calling the necessary elaboration routines. A call to `adainit` is required before the first call to an Ada subprogram.

`adafinal` You must call this routine to perform any library-level finalization required by the Ada subprograms. A call to `adafinal` is required after the last call to an Ada subprogram, and before the program terminates.

If the `-n` switch is given, more than one ALI file may appear on the command line for `gnatbind`. The normal *closure* calculation is performed for each of the specified units. Calculating the closure means finding out the set of units involved by tracing `with` references. The reason it is necessary to be able to specify more than one ALI file is that a given program may invoke two or more quite separate groups of Ada units.

The binder takes the name of its output file from the first specified ALI file, unless overridden by the use of the `-o file`. The output file is a C source file, which must be compiled using the C compiler. This compilation occurs automatically as part of the `gnatmake` processing.

4.7 Summary of Binder Switches

The following are the switches available with `gnatbind`:

`-b` Generate brief messages to `stderr` even if verbose mode set.

`-c` Check only, no generation of binder output file.

`-e` Output complete list of elaboration-order dependencies.

`-aI` Specify directory to be searched for source file.

`-aO` Specify directory to be searched for ALI files.

`-I` Specify directory to be searched for source and ALI files.

`-I-` Do not look for sources in the current directory where `gnatbind` was invoked, and do not look for ALI files in the directory containing the ALI file named in the `gnatbind` command line.

`-l` Output chosen elaboration order.

`-m`*n* Limit number of detected errors to *n* (1-999).

`-n` No main program.

`-o` *file* Name the output file *file* (default is '`b_`*xxx*`.c`').

`-s` Require all source files to be present.

`-t` Ignore time-stamp errors.

`-v` Verbose mode. Write error messages, header, summary output to `stdout`.

`-w`*x* Warning mode (*x*=s/e for suppress/treat as error)

`-x` Exclude source files (check object consistency only).

You may obtain this listing by running the program `gnatbind` with no arguments.

4.8 Command-Line Access

The package Ada.Command_Line provides access to the command-line arguments and program name. In order for this interface to operate correctly, the two variables

```
int gnat_argc;
char **gnat_argv;
```

are declared in one of the GNAT library routines. These variables must be set from the actual argc and argv values passed to the main program. With no n present, gnatbind generates the C main program to automatically set these variables. If the n switch is used, there is no automatic way to set these variables. If they are not set, the procedures in Ada.Command_Line will not be available, and any attempt to use them will raise Constraint_Error. If command line access is required, your main program must set gnat_argc and gnat_argv from the argc and argv values passed to it.

4.9 Search Paths for gnatbind

The binder takes the name of an ALI file as its argument and needs to locate source files as well as other ALI files to verify object consistency.

For source files, it follows exactly the same search rules as gcc (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33). For ALI files the directories searched are:

1. The directory containing the ALI file named in the command line, unless the switch -I- is specified.
2. All directories specified by -I switches on the gnatbind command line, in the order given.
3. Each of the directories listed in the value of the ADA_OBJECTS_PATH environment variable. Construct this value exactly as the PATH environment variable: a list of directory names separated by colons.
4. The default location for the GNAT Run-Time Library (RTL) files, determined when GNAT was built and installed on your system.

In the binder the switch -I is used to specify both source and library file paths. Use -aI instead if you want to specify source paths only, and -aO if you want to specify libary paths only. This means that for the binder -I*dir* is equivalent to -aI*dir* -aO*dir*. The binder generates the bind file (a C language source file) in the current working directory.

The packages Ada, System, and Interfaces and their children make up the GNAT Run-Time Library, together with the package GNAT and its children, which contain a set of useful additional library functions provided by GNAT. The sources for these units are needed by the compiler and are kept together in one directory. The ALI files and object files generated by compiling the RTL are needed by the binder and the linker and are kept together in one directory, typically different from the directory containing the sources. In a normal installation, you need not specify these directory names when compiling or binding. Either the environment variables or the built-in defaults cause these files to be found.

Besides simplifying access to the RTL, a major use of search paths is in compiling sources from multiple directories. This can make development environments much more flexible.

4.10 Examples of gnatbind Usage

This section contains a number of examples of using the GNAT binding utility `gnatbind`.

`gnatbind hello.ali`

The main program `Hello` (source program in '`hello.adb`') is bound using the standard switch settings. The generated main program is '`b_hello.c`'. This is the normal, default use of the binder.

`gnatbind main.ali -o mainprog.c -x -e`

The main program `Main` (source program in '`main.adb`') is bound, excluding source files from the consistency checking, generating the file '`mainprog.c`'.

`gnatbind -x main_program.ali -o mainprog.c`

This command is exactly the same as the previous example. Switches may appear anywhere in the command line, and single letter switches may be combined into a single switch.

`gnatbind -n math.ali dbase.ali -o ada-control.c`

The main program is in a language other than Ada, but calls to subprograms in packages `Math` and `Dbase` appear. This call to `gnatbind` generates the file '`control.c`' containing the `adainit` and `adafinal` routines to be called before and after accessing the Ada units.

5 Linking Using gnatlink

This chapter discusses `gnatlink`, a utility program used to link Ada programs and build an executable file. This is a simple program that invokes the UNIX linker (via the `gcc` command) with a correct list of object files and library references. `gnatlink` automatically determines the list of files and references for the Ada part of a program. It uses the binder file generated by the binder to determine this list.

5.1 Running gnatlink

The form of the `gnatlink` command is

`gnatlink [`*switches*`]` *mainprog*`[.ali] [`*non-Ada objects*`] [`*linker options*`]`

'*mainprog*`.ali`' references the ALI file of the main program. The '`.ali`' extension of this file can be omitted. From this reference, `gnatlink` locates the corresponding binder file '`b_`*mainprog*`.c`' and, using the information in this file along with the list of non-Ada objects and linker options, constructs a UNIX linker command file to create the executable.

The arguments following '*mainprog*`.ali`' are passed to the linker uninterpreted. They typically include the names of object files for units written in other languages than Ada and any library references required to resolve references in any of these foreign language units, or in `pragma Import` statements in any Ada units. This list may also include linker switches.

`gnatlink` determines the list of objects required by the Ada program and prepends them to the list of objects passed to the linker. `gnatlink` also gathers any arguments set by the use of `pragma Linker_Options` and adds them to the list of arguments presented to the linker.

5.2 Switches for gnatlink

The following switches are available with the `gnatlink` utility:

`--GCC=<string>`
: Program for compiling binder file; default gcc'. Quotes around `<string>` are optional if `<string>` contains no spaces or other separator characters.

`-o` *exec-name*
: *exec-name* specifies an alternate name for the generated executable program. If this switch is omitted, the executable has the same name as the main unit. For example, `gnatlink try.ali` creates an executable called '`try`'.

`-v`
: Causes additional information to be output, including a full list of the included object files. This switch option is most useful when you want to see what set of object files are being used in the link step.

`-g`
: The option to include debugging information causes the C bind file (in other words, '`b_`*mainprog*`.c`') to be compiled with `-g`. In addition, the binder does not delete the '`b_`*mainprog*`.c`' and '`b_`*mainprog*`.o`' files. Without `-g`, the binder removes these files by default.

`-gnatlink` *name*
: *name* is the name of the linker to be invoked. You normally omit this switch, in which case the default name for the linker is (‘`gcc`’).

6 The GNAT Make Program gnatmake

A typical development cycle when working on an Ada program consists of the following steps:

1. Edit some sources to fix bugs.
2. Add enhancements.
3. Compile all sources affected.
4. Rebind and relink.
5. Test.

The third step can be tricky, because not only do the modified files have to be compiled, but any files depending on these files must also be recompiled. The dependency rules in Ada can be quite complex, especially in the presence of overloading, `use` clauses, generics and inlined subprograms.

`gnatmake` automatically takes care of the third and fourth steps of this process. It determines which sources need to be compiled, compiles them, and binds and links the resulting object files.

Unlike some other Ada make programs, the dependencies are always accurately recomputed from the new sources. The source based approach of the GNAT compilation model makes this possible. This means that if changes to the source program cause corresponding changes in dependencies, they will always be tracked exactly correctly by `gnatmake`.

6.1 Running gnatmake

The `gnatmake` command has the form

```
$ gnatmake switches file_name mode_switches
```

The only required argument is *file_name*, which specifies the compilation unit that is the main program. If `switches` are present, they can be placed before of after *file_name*. If *mode_switches* are present, they must always be placed after *file_name* and all `switches`.

If you are using standard file extensions (.adb and .ads), then the extension may be omitted from the *file_name* argument. However, if you are using non-standard extensions, then it is required that the extension be given. A relative or absolute directory path can be specified in *file_name*, in which case, the input source file will be searched for in the specified directory only. Otherwise, the input source file will first be searched in the directory where `gnatmake` was invoked and if it is not found, it will be search on the source path of the compiler as described in Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33.

All `gnatmake` output (except when you specifiy `-M`) is to `stderr`. The output produced by the `-M` switch is send to `stdout`.

6.2 Switches for `gnatmake`

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

`--GCC=<string>`
: Program for compiling; default `gcc`'. Quotes around `<string>` are optional if `<string>` contains no spaces or other separator characters. They are necessary if you want to pass special switches to your compiler program (for instance `--GCC="foo -x -y"`). Note that, for the time being, switch `-c` is always inserted after you command name (thus in the above example the compiler command that will be generated by `gnatmake` will be `foo -c -x- y`).

`--GNATBIND=<string>`
: Program for binding; default `gnatbind`'. Quotes around `<string>` are optional if `<string>` contains no spaces or other separator characters. As an example `--GNATBIND="bar -x -y"` will instruct `gnatmake` to use `bar -x -y` as your binder. Binder switches that are normally appended by `gnatmake` to `gnatbind`' are now appended to the end of `bar -x -y`.

`--GNATLINK=<string>`
: Program for linking; default `gnatlink`'. Quotes around `<string>` are optional if `<string>` contains no spaces or other separator characters. As an example `--GNATLINK="lan -x -y"` will instruct `gnatmake` to use `man -x -y` as your linker. Linker switches that are normally appended by `gnatmake` to `gnatlink`' are now appended to the end of `lan -x -y`.

`-a`
: Consider all files in the make process, even the GNAT internal system files (for example, the predefined Ada library files), as well as any locked files. Locked files are files whose ALI file is write-protected. By default, `gnatmake` does not check these files, because the assumption is that the GNAT internal files are properly up to date, and also that any write protected ALI files have been properly installed. Note that if there is an installation problem, such that one of these files is not up to date, it will be properly caught by the binder. You may have to specify this switch if you are working on GNAT itself. `-f` is also useful in conjunction with `-f` if you need to recompile an entire application, including run-time files, using special configuration pragma settings, such as a non-standard `Float_Representation` pragma. By default `gnatmake -a` compiles all GNAT internal files with `gcc -c -gnatg` rather than `gcc -c`.

`-c`
: Compile only. Do not perform binding and linking. If the root unit specified by *file_name* is not a main unit, this is the default. Otherwise `gnatmake` will attempt binding and linking unless all objects are up to date and the executable is more recent than the objects.

`-f`
: Force recompilations. Recompile all sources, even though some object files may be up to date, but don't recompile predefined or GNAT internal files or locked files (files with a write-protected ALI file), unless the `-a` switch is also specified.

`-j`*n*
: Use *n* processes to carry out the (re)complations. On a multiprocessor machine compilations will occur in parallel. In the event of compilation errors, messages

from various compilations might get interspersed (but `gnatmake` will give you the full ordered list of failing compiles at the end). If this is problematic, rerun the make process with n set to 1 to get a clean list of messages.

`-k` Keep going. Continue as much as possible after a compilation error. To ease the programmer's task in case of compilation errors, the list of sources for which the compile fails is given when `gnatmake` terminates.

`-M` Check if all objects are up to date. If they are, output the object dependences to `stdout` in a form that can be directly exploited in a '`Makefile`'. By default, each source file is prefixed with its (relative or absolute) directory name. This name is whatever you specified in the various `-aI` and `-I` switches. If you use `gnatmake -M -q` (see below), only the source file names, without relative paths, are output. If you just specify the `-M` switch, dependencies of the GNAT internal system files are omitted. This is typically what you want. If you also specify the `-a` switch, dependencies of the GNAT internal files are also listed. Note that dependencies of the objects in external Ada libraries (see switch `-aL`*dir* in the following list) are never reported.

`-i` In normal mode, `gnatmake` compiles all object files and ALI files into the current directory. If the `-i` switch is used, then instead object files and ALI files that already exist are overwritten in place. This means that once a large project is organized into separate directories in the desired manner, then `gnatmake` will automatically maintain and update this organization. If no ALI files are found on the Ada object path (Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33), the new object and ALI files are created in the directory containing the source being compiled. If another organization is desired, where objects and sources are kept in different directories, a useful technique is to create dummy ALI files in the desired directories. When detecting such a dummy file, `gnatmake` will be forced to recompile the corresponding source file, and it will be put the resulting object and ALI files in the directory where it found the dummy file.

`-m` Specifies that the minimum necessary amount of recompilation be performed. In this mode (`gnatmake`) ignores time stamp differences when the only modifications to a source file consist in adding/removing comments, empty lines, spaces or tabs. This means that if you have changed the comments in a source file or have simply reformatted it, using this switch will tell gnatmake not to recompile files that depend on it (provided other sources on which these files depend have undergone no semantic modifications).

`-n` Don't compile, bind, or link. Checks if all objects are up to date. If they are not, the full name of the first file that needs to be recompiled is printed. Repeated use of this option, followed by compiling the indicated source file, will eventually result in recompiling all required units.

`-o` *exec˙name*

Output euecutable name. The name of the final executable program will be *exec_name*. If the `-o` switch is omitted the default name for the executable will be the name of the input file in appropriate form for an executable file on the host system.

`-q` Quiet. When this flag is not set, the commands carried out by `gnatmake` are displayed.

`-v` Verbose. Displays the reason for all recompilations `gnatmake` decides are necessary.

`gcc switches`
The switch `-g` or any uppercase switch (other than `-A`, or `-L`) or any switch that is more than one character is passed to `gcc` (e.g. `-O`, `-gnato,` etc.)

Source and library search path switches:

`-aI`*dir* When looking for source files also look in directory *dir*. The order in which source files search is undertaken is described in Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 33.

`-aL`*dir* Consider *dir* as being an externally provided Ada library. Instructs `gnatmake` to skip compilation units whose '`.ali`' files have been located in directory *dir*. This allows you to have missing bodies for the units in *dir*. You still need to specify the location of the specs for these units by using the switches `-aI`*dir* or `-I`*dir*. Note: this switch is provided for compatibility with previous versions of `gnatmake`. The easier method of causing standard libraries to be excluded from consideration is to write-protect the corresponding ALI files.

`-aO`*dir* When searching for library and object files, look in directory *dir*. The order in which library files are searched is described in Section 4.9 [Search Paths for gnatbind], page 41.

`-A`*dir* Equivalent to `-aL`*dir* `-aI`*dir*.

`-I`*dir* Equivalent to `-aO`*dir* `-aI`*dir*.

`-I-` Do not look for source files in the directory containing the source file named in the command line. Do not look for ALI or object files in the directory where `gnatmake` was invoked.

`-L`*dir* Add directory *dir* to the list of directories in which the linker will search for libraries. This is equivalent to `-largs -L`*dir*.

6.3 Mode switches for `gnatmake`

The mode switches (refferred to as `mode_switches`) allow the inclusion of switches that are to be passed to the compiler itself, the binder or the linker. The effect of a mode switch is to cause all subsequent switches up to the end of the switch list, or up to the next mode switch, to be interpreted as switches to be passed on to the designated component of GNAT.

`-cargs` *switches*
Compiler switches. Here *switches* is a list of switches that are valid switches for `gcc`. They will be passed on to all compile steps performed by `gnatmake`.

`-bargs` *switches*
Binder switches. Here *switches* is a list of switches that are valid switches for `gcc`. They will be passed on to all bind steps performed by `gnatmake`.

`-largs` *switches*
Linker switches. Here *switches* is a list of switches that are valid switches for `gcc`. They will be passed on to all link steps performed by `gnatmake`.

6.4 Notes on the Command Line

This section contains some additional useful notes on the operation of the `gnatmake` command.

- If `gnatmake` finds no ALI files, it recompiles the main program and all other units required by the main program. This means that `gnatmake` can be used for the initial compile, as well as during subsequent steps of the development cycle.
- If you enter `gnatmake` *file*`.adb`, where '*file*`.adb`' is a subunit or body of a generic unit, `gnatmake` recompiles '*file*`.adb`' (because it finds no ALI) and stops, issuing a warning.
- In `gnatmake` the switch `-I` is used to specify both source and library file paths. Use `-aI` instead if you just want to specify source paths only and `-aO` if you want to specify libary paths only.
- `gnatmake` examines both an ALI file and its corresponding object file for consistency. If an ALI is more recent than its corresponding object, or if the object file is missing, the corresponding source will be recompiled. Note that `gnatmake` expects an ALI and the corresponding object file to be in the same directory.
- `gnatmake` will ignore any files whose ALI file is write-protected. This may conveniently be used to exclude standard libraries from consideration and in particular it means that the use of the `-f` switch will not recompile these files unless `-a` is also specified.
- `gnatmake` has been designed to make the use of Ada libraries particularly convenient. Assume you have an Ada library organized as follows: *obj-dir* contains the objects and ALI files for of your Ada compilation units, whereas *include-dir* contains the specs of these units, but no bodies. Then to compile a unit stored in `main.adb`, which uses this Ada library you would just type

  ```
  $ gnatmake -aIinclude-dir  -aLobj-dir  main
  ```

- Using `gnatmake` along with the `-m (minimal recompilation)` switch provides an extremely powerful tool: you can freely update the comments/format of your source files without having to recompile everything. Note, however, that adding or deleting lines in a source files may render its debugging info obsolete. If the file in question is a spec, the impact is rather limited, as that debugging info will only be useful during the elaboration phase of your program. For bodies the impact can be more significant. In all events, your debugger will warn you if a source file is more recent than the corresponding object, and therefore obsolescence of debugging information will go unnoticed.

6.5 How gnatmake Works

Generally `gnatmake` automatically performs all necessary recompilations and you don't need to worry about how it works. However, it may be useful to have some basic understanding of the `gnatmake` approach and in particular to understand how it uses the results of previous compilations without incorrectly depending on them.

First a definition: an object file is considered *up to date* if the corresponding ALI file exists and its time stamp predates that of the object file and if all the source files listed in the dependency section of this ALI file have time stamps matching those in the ALI file. This means that neither the source file itself nor any files that it depends on have been modified, and hence there is no need to recompile this file.

`gnatmake` works by first checking if the specified main unit is up to date. If so, no compilations are required for the main unit. If not, `gnatmake` compiles the main program to build a new ALI file that reflects the latest sources. Then the ALI file of the main unit is examined to find all the source files on which the main program depends, and `gnatmake` recursively applies the above procedure on all these files.

This process ensures that `gnatmake` only trusts the dependencies in an existing ALI file if they are known to be correct. Otherwise it always recompiles to determine a new, guaranteed accurate set of dependencies. As a result the program is compiled "upside down" from what may be more familiar as the required order of compilation in some other Ada systems. In particular, clients are compiled before the units on which they depend. The ability of GNAT to compile in any order is critical in allowing an order of compilation to be chosen that guarantees that `gnatmake` will recompute a correct set of new dependencies if necessary.

6.6 Examples of gnatmake Usage

`gnatmake hello.adb`

Compile all files necessary to bind and link the main program '`hello.adb`' (containing unit `Hello`) and bind and link the resulting object files to generate an executable file '`hello`'.

`gnatmake -q Main_Unit -cargs -O2 -bargs -l`

Compile all files necessary to bind and link the main program unit `Main_Unit` (from file '`main_unit.adb`'). All compilations will be done with optimization level 2 and the order of elaboration will be listed by the binder. `gnatmake` will operate in quiet mode, not displaying commands it is executing.

7 Renaming Files Using gnatchop

This chapter discusses how to handle files with multiple units by using the gnatchop utility. This utility is also useful in renaming files to meet the standard GNAT default file naming conventions.

7.1 Handling Files with Multiple Units

The basic compilation model of GNAT requires that a file submitted to the compiler have only one unit and there be a strict correspondence between the file name and the unit name.

The gnatchop utility allows both of these rules to be relaxed, allowing GNAT to process files which contain multiple compilation units and files with arbitrary file names. gnatchop reads the specified file and generates one or more output files, containing one unit per file. The unit and the file name correspond, as required by GNAT.

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 and work with them from that point on.

Alternatively, if you want to keep your files in the "foreign" format, perhaps to maintain compatibility with some other Ada compilation system, you can set up a procedure where you use gnatchop each time you compile, regarding the source files that it writes as temporary files that you throw away.

7.2 Operation 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 files 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 needing to mark it explicitly and most users will find this default to be what they want. In this default mode it is incorrect to submit a file containing only configuration pragmas, or one that ends in configuration pragmas, to gnatchop.

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

First, in compilation mode, if gnatchop is given a file that consists of only configuration pragmas, then this file is appended 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 the section on handling of configuration pragmas see Chapter 8 [Handling of Configuration Pragmas], page 55.

Second, in compilation mode, if gnatchop is given a file that starts with configuration pragmas, and contains one or more units, then these configuration pragmas are prepended to each of the chopped files. This behavior provides the required behavior descfribed 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, gnatchop will be used in default mode. The compilation mode described above is used only if you need exactly 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, and is for example the mode in which GNAT processes the ACVC tests.

7.3 Command Line for gnatchop

The gnatchop command has the form:

```
$ gnatchop switches file name [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 it is not specified, all files are written to the current directory.

For example, given a file called 'hellofiles' containing

```
procedure hello;

with Text_IO; use 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. The generated files can be compiled in the normal manner.

7.4 Switches for gnatchop

gnatchop recognizes the following switches:

-c Causes gnatchop to operate in compilation mode, in which configuration pragmas are handled according to strict RM rules. See previous section for a full description of this mode.

-gnatxxx This passes the given -gnatxxx switch to gnat which is used to parse the given file. Not all xxx options make sense, but for example, the use of -gnati2 allows gnatchop to process a source file that uses Latin-2 coding for identifiers.

-h Causes gnatchop to generate a brief help summary to the standard output file showing usage information.

-k *mmmm*

Limit generated file names to the specified number mm of characters. This is useful if the resulting set of files is required to be interoperable with systems which limit the length of file names. No space is allowed between the -k and the numeric value. The numeric value may be omitted in which case a default of -k8, suitable for use with DOS-like file systems, is used. If no -k switch is present then there is no limit on the length of file names.

-q Causes output of informational messages indicating the set of generated files to be suppressed. Warnings and error messages are unaffected.

-r Generate Source_Reference pragmas. Use this switch if the output files are regarded as temporary and development is to be done in terms of the original unchopped file. This switch causes Source_Reference pragmas to be inserted into each of the generated files to refers back to the original file name and line number. The result is that all error messages refer back to the original unchopped file. In addition, the debugging information placed into the object file (when the -g switch of gcc or gnatmake is specified) also refers back to this original file so that tools like profilers and debuggers will give information in terms of the original unchopped file.

If the original file to be chopped itself contains a Source_Reference pragma referencing a third file, then gnatchop respects this pragma, and the generated Source_Reference pragmas in the chopped file refer to the original file, with appropriate line numbers. This is particularly useful when gnatchop is used in conjunction with gnatprep to compile files that contain preprocessing statements and multiple units.

-v Causes gnatchop to operate in verbose mode. The version number and copyright notice are output, as well as exact copies of the gnat1 commands spawned to obtain the chop control information.

-w Overwrite existing file names. Normally gnatchop regards it as a fatal error if there is already a file with the same name as a file it would otherwise output, in otherwords if the files to be chopped contain duplicated units. This switch bypasses this check, and causes all but the last instance of such duplicated units to be skipped.

7.5 Examples of `gnatchop` Usage

`gnatchop -w hello_s.ada ichbiah/files`

Chops the source file '`hello_s.ada`'. The output files will be placed in the directory '`ichbiah/files`', overwriting any files with matching names in that directory (no files in the current directory are modified).

`gnatchop archive`

Chops the source file '`archive`' into the current directory. One useful application of `gnatchop` is in sending sets of sources around, for example in email messages. The required sources are simply concatenated (for example, using a UNIX `cat` command), and then `gnatchop` is used at the other end to reconstitute the original file names.

`gnatchop file1 file2 file3 direc`

Chops all units in files '`file1`', '`file2`', '`file3`', placing the resulting files in the directory '`direc`'. Note that if any units occur more than once anywhere within this set of files, an error message is generated, and no files are written. To override this check, use the `-w` switch, in which case the last occurrence in the last file will be the one that is output, and earlier duplicate occurrences for a given unit will be skipped.

8 Handling of Configuration Pragmas

In Ada 95, configuration pragmas include those pragmas described as such in the Ada 95 Reference Manual, as well as implementation-dependent pragmas that are configuration pragmas. See the individual descriptions of pragmas 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.

Configuration pragmas may either appear at the start of a compilation unit, in which case they apply only to that unit, or they may apply to all compilations performed in a given compilation environment.

GNAT also provides the `gnatchop` utility to provide an automatic way to handle configuration pragmas following the semantics for compilations (that is, files with multiple units), described in the RM. See section see Section 7.2 [Operating gnatchop in Compilation Mode], page 51 for details. However, for most purposes, it will be more convenient to edit the '`gnat.adc`' file that contains configuration pragmas directly, as described in the following section.

8.1 The `gnat.adc` file

In GNAT a compilation environment is defined by the current directory at the time that a compile command is given. This current directory is searched for a file whose name is '`gnat.adc`'. If this file is present, it is expected to contain one or more configuration pragmas that will be applied to the current compilation.

Configuration pragmas may be entered into the '`gnat.adc`' file either by running `gnatchop` on a source file that consists only of configuration pragmas, or more conveniently by direct editing of the '`gnat.adc`' file, which is a standard format source file.

9 Elaboration Order Handling in GNAT

This chapter describes the handling of elaboration code in Ada 95 and in GNAT, and discusses how the order of elaboration of program units can be controlled in GNAT, either automatically or with explicit programming features.

9.1 Elaboration Code in Ada 95

Ada 95 provides rather general mechanisms for executing code at elaboration time, that is to say before the main program starts executing. Such code arises in three contexts:

Initializers for variables.

Variables declared at the library level, in package specs or bodies, can require initialization that is performed at elaboration time, as in:

```
Sqrt_Half : Float := Sqrt (0.5);
```

Package initialization code

Code in a `BEGIN-END` section at the outer level of a package body is executed as part of the package body elaboration code.

Library level task allocators

Tasks that are declared using task allocators at the library level start executing immediately and hence can execute at elaboration time.

Subprogram calls are possible in any of these contexts, which means that any arbitrary part of the program may be executed as part of the elaboration code. It is even possible to write a program which does all its work at elaboration time, with a null main program, although stylistically this would usually be considered an inappropriate way to structure a program.

An important concern arises in the context of elaboration code: we have to be sure that it is executed in an appropriate order. What we have is numerous sections of elaboration code, potentially one section for each unit in the program. It is important that these execute in the correct order. Correctness here means that, taking the above example of the declaration of `Sqrt_Half`, that if some other piece of elaboration code references `Sqrt_Half`, then it must run after the section of elaboration code that contains the declaration of `Sqrt_Half`.

There would never be any order of elaboration problem if we made a rule that whenever you `with` a unit, you must elaborate both the spec and body of that unit before elaborating the unit doing the `with`'ing:

```
with Unit_1;
package Unit_2 is ...
```

would require that both the body and spec of `Unit_1` be elaborated before the spec of `Unit_2`. However, a rule like that would be far too restrictive. In particular, it would make it impossible to have routines in separate packages that were mutually recursive.

You might think that a clever enough compiler could look at the actual elaboration code and determine an appropriate correct order of elaboration, but in the general case, this is not possible. Consider the following example.

In the body of `Unit_1`, we have a procedure `Func_1` that references the variable `Sqrt_1`, which is declared in the elaboration code of the body of `Unit_1`:

```
     Sqrt_1 : Float := Sqrt (0.1);
```

The elaboration code of the body of `Unit_1` also contains:

```
     if expression_1 = 1 then
        Q := Unit_2.Func_2;
     end if;
```

`Unit_2` is exactly parallel, it has a procedure `Func_2` that references the variable `Sqrt_2`, which is declared in the elaboration code of the body `Unit_2`:

```
     Sqrt_2 : Float := Sqrt (0.1);
```

The elaboration code of the body of `Unit_2` also contains:

```
     if expression_2 = 2 then
        Q := Unit_1.Func_1;
     end if;
```

Now the question is, which of the following orders of elaboration is acceptable:

```
     Spec of Unit_1
     Spec of Unit_2
     Body of Unit_1
     Body of Unit_2
```

or

```
     Spec of Unit_2
     Spec of Unit_1
     Body of Unit_2
     Body of Unit_1
```

If you carefully analyze the flow here, you will see that you cannot tell at compile time the answer to this question. If `expression_1` is not equal to 1, and `expression_2` is not equal to 2, then either order is acceptable, because neither of the function calls is executed. If both tests evaluate to true, then neither order is acceptable and in fact there is no correct order.

If one of the two expressions is true, and the other is false, then one of the above orders is correct, and the other is incorrect. For example, if `expression_1` = 1 and `expression_2` /= 2, then the call to `Func_2` will occur, but not the call to `Func_1.` This means that it is essential to elaborate the body of `Unit_1` before the body of `Unit_2`, so the first order of elaboration is correct and the second is wrong.

By making `expression_1` and `expression_2` depend on input data, or perhaps the time of day, we can make it impossible for the compiler or binder to figure out which of these expressions will be true, and hence it is impossible to guarantee a safe order of elaboration at run time.

9.2 Checking the Elaboration Order in Ada 95

In some languages that involve the same kind of elaboration problems, e.g. Java and C++, the programmer is expected to worry about these ordering problems himself, and it is common to write a program in which an incorrect elaboration order gives surprising results, because it references variables before they are initialized. Ada 95 is designed to be a safe language, and a programmer-beware approach is clearly not sufficient. Consequently, the language provides three lines of defense:

Standard rules

Some standard rules restrict the possible choice of elaboration order. In particular, if you `with` a unit, then its spec is always elaborated before the unit doing the `with`. Similarly, a parent spec is always elaborated before the child spec, and finally a spec is always elaborated before its corresponding body.

Dynamic elaboration checks

Dynamic checks are made at run time, so that if some entity is accessed before it is elaborated (typically by means of a subprogram call) then the exception (`Program_Error`) is raised.

Elaboration control

Facilities are provided for the programmer to specify the desired order of elaboration.

Let's look at these facilities in more detail. First, the rules for dynamic checking. One possible rule would be simply to say that the exception is raised if you access a variable which has not yet been elaborated. The trouble with this approach is that it could require expensive checks on every variable reference. Instead Ada 95 has two rules which are a little more restrictive, but easier to check, and easier to state:

Restrictions on calls

A subprogram can only be called at elaboration time if its body has been elaborated. The rules for elaboration given above guarantee that the spec of the subprogram has been elaborated before the call, but not the body. If this rule is violated, then the exception `Program_Error` is raised.

Restrictions on instantiations

A generic unit can only be instantiated if the body of the generic unit has been elaborated. Again, the rules for elaboration given above guarantee that the spec of the generic unit has been elaborated before the instantiation, but not the body. if this rule is violated, then the exception `Program_Error` is raised.

The idea is that if the body has been elaborated, then any variables it references must have been elaborated; by checking for the body being elaborated we guarantee that none of its references causes any trouble. As we noted above, this is a little too restrictive, because a subprogram that has no non-local references in its body may in fact be safe to call. However, it really would be unsafe to rely on this, because it would mean that the caller was aware of details of the implementation in the body. This goes agains the basic tenets of Ada.

A plausible implementation can be described as follows. A Boolean variable is associated with each subprogram and each generic unit. This variable is initialized to False, and is set to True at the point body is elaborated. Every call or instantiation checks the variable, and raises `Program_Error` if the variable is False.

9.3 Controlling the Elaboration Order in Ada 95

In the previous section we discussed the rules in Ada 95 which ensure that `Program_Error` is raised if an incorrect elaboration order is chosen. This prevents erroneous executions, but we need mechanisms to specify a correct execution and avoid the exception altogether. To achieve this, Ada 95 provides a number of features for controlling the order of elaboration. We discuss these features in this section.

First, there are several ways of indicating to the compiler that a given unit has no elaboration problems:

packages that do not require a body

In Ada 95, a library package that does not require a body does not permit a body. This means that if we have a such a package, as in:

```
package Definitions is
   generic
      type m is new integer;
   package Subp is
      type a is array (1 .. 10) of m;
      type b is array (1 .. 20) of m;
   end Subp;
end Definitions;
```

A package that `with`'s `Definitions` may safely instantiate `Definitions.Subp` because the compiler can determine that there definitely is no package body to worry about in this case

pragma Pure

Places sufficient restrictions on a unit to guarantee that no call to any subprogram in the unit can result in an elaboration problem. This means that the compiler does not need to worry about the point of elaboration of such units, and in particular, does not need to check any calls to any subprograms in this unit.

pragma Preelaborate

This pragma places slightly less stringent restrictions on a unit than does pragma Pure, but these restrictions are still sufficient to ensure that there are no elaboration problems with any calls to the unit.

pragma Elaborate_Body

This pragma requires that the body of a unit be elaborated immediately after its spec. Suppose a unit `A` has such a pragma, and unit `B` does a `with` of unit `A`. Recall that the standard rules require the spec of unit `A` to be elaborated before the `with`'ing unit; given the pragma in `A`, we also know that the body of `A` will be elaborated before `B`, so that calls to `A` are safe and do not need a check.

Note that, unlike pragma `Pure` and pragma `Preelaborate`, the use of `Elaborate_Body` does not guarantee that the program is free of elaboration problems, because it may not be possible to satisfy the requested elaboration order. Let's go back to the example with `Unit_1` and `Unit_2`. If a programmer marks `Unit_1` as `Elaborate_Body`, and not `Unit_2,` then the order of elaboration will be:

```
Spec of Unit_2
Spec of Unit_1
Body of Unit_1
Body of Unit_2
```

Now that means that the call to `Func_1` in `Unit_2` need not be checked, it must be safe. But the call to `Func_2` in `Unit_1` may still fail if `Expression_1` is equal to 1, and the programmer must still take responsibility for this not being the case.

If all units carry a pragma `Elaborate_Body`, then all problems are eliminated, except for calls entirely within a body, which are in any case fully under programmer control. However, using the pragma everywhere is not always possible. In particular, for our `Unit_1`/`Unit_2` example, if we marked both of them as having pragma `Elaborate_Body`, then clearly there would be no possible elaboration order.

The above pragmas allow a server to guarantee safe use by clients, and clearly this is the preferable approach. Consequently a good rule in Ada 95 is to mark units as `Pure` or `Preelaborate` if possible, and if this is not possible, mark them as `Elaborate_Body` if possible. As we have seen, there are situation where neither of these three pragmas can be used. So we also provide methods for clients to control the order of elaboration of the servers on which they depend:

pragma Elaborate (unit)
: This pragma is placed in the context clause, after a `with` statement, and it requires that the body of the named unit be elaborated before the unit in which the pragma occurs. The idea is to use this pragma if the current unit calls at elaboaration time, directly or indirectly, some subprogram in the named unit.

pragma Elaborate_All (unit)
: This is a stronger version of the Elaborate pragma. Consider the following example:

```
Unit A with's unit B and calls B.Func in elaboration code
Unit B with's unit C, and B.Func calls C.Func
```

Now if we put a pragma `Elaborate (B)` in unit `A`, this ensures that the body of `B` is elaborated before the call, but not the body of `C`, so the call to `C.Func` could still cause `Program_Error` to be raised.

The effect of a pragma `Elaborate_All` is stronger, it requires not only that the body of the named unit be elaborated before the unit doing the `with`, but also the bodies of all units that the named unit uses, following `with` links transitively. For example, if we put a pragma `Elaborate_All (B)` in unit `A`, then it requires not only that the body of `B` be elaborated before `A`, but also the body of `C`, because `B` `with`'s `C`.

We are now in a position to give a usage rule in Ada 95 for avoiding elaboration problems, at least if dynamic dispatching and access to subprogram values are not used. We will handle these cases separately later.

The rule is simple. If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a `with`'ed unit, or instantiate a generic unit in a `with`'ed unit, then if the `with`'ed unit does not have pragma Pure, Preelaborate, or Elaborate_Body, then the client should have an Elaborate_All for the `with`'ed unit. By following this rule a client is assured that calls can be made without risk of an exception. If this rule is not followed, then a program may be in one of four states:

No order exists

No order of elaboration exists which follows the rules, taking into account any Elaborate, Elaborate_All, or Elaborate_Body pragmas. In this case, an Ada 95 compiler must diagnose the situation at bind time, and refuse to build an executable program.

One or more orders exist, all incorrect

One or more acceptable elaboration orders exists, and all of them generate an elaboration order problem. In this case, the binder can build an executable program, but Program_Error will be raised when the program is run.

Several orders exist, some right, some incorrect

One or more acceptable elaboration orders exists, and some of them work, and some do not. The programmer has not controlled the order of elaboration, so the binder may or may not pick one of the correct orders, and the program may or may not raise an exception when it is run. This is the worst case, because it means that the program may fail when moved to another compiler, or even another version of the same compiler.

One or more orders exists, all correct

One ore more acceptale elaboration orders exist, and all of them work. In this case the program runs successfully. This state of affairs can be guaranteed by following the rule we gave above, but may be true even if the rule is not followed.

Note that one additional advantage of following our Elaborate_All rule is that the program continues to stay in the ideal (all orders OK) state even if maintenance changes some bodies of some subprograms. Conversely, if a program that does not follow this rule happens to be safe at some point, this state of affairs may deteriorate silently as a result of maintenance changes.

9.4 Controlling Elaboration in GNAT - Internal Calls

In the case of internal calls, i.e. calls within a single package, the programmer has full control over the order of elaboration, and it is up to the programmer to elaborate declarations in an appropriate order. For example writing:

```
function One return Float;

Q : Float := One;

function One return Float is
begin
     return 1.0;
end One;
```

will obviously raise Program_Error at run time, because function One will be called before its body is elaborated. In this case GNAT will generate a warning that the call will raise Program_Error:

```
 1. procedure y is
 2.    function One return Float;
 3.
 4.    Q : Float := One;
                    |
    >>> warning: cannot call "One" before body is elaborated
    >>> warning: Program_Error will be raised at run time

 5.
 6.    function One return Float is
 7.    begin
 8.         return 1.0;
 9.    end One;
 0.
11. begin
12.    null;
13. end;
```

Note that in this particular case, it is likely that the call is safe, because the function `One` does not access any global variables. Nevertheless in Ada 95, we do not want the validity of the check to depend on the contents of the body (think about the separate compilation case), so this is still wrong, as we discussed in the previous sections.

The error is easily corrected by rearranging the declarations so that the body of One appears before the declaration containing the call (note that in Ada 95, declarations can appear in any order, so there is no restriction that would prevent this reordering, and if we write:

```
function One return Float;

function One return Float is
begin
     return 1.0;
end One;

Q : Float := One;
```

then all is well, no warning is generated, and no `Program_Error` exception will be raised. Things are more complicated when a chain of subprograms is executed:

```
function A return Integer;
function B return Integer;
function C return Integer;

function B return Integer is begin return A; end;
function C return Integer is begin return B; end;

X : Integer := C;

function A return Integer is begin return 1; end;
```

Now the call to `C` at elaboration time in the declaration of `X` is correct, because the body of `C` is already elaborated, and the call to `B` within the body of `C` is correct, but the call to `A` within the body of `B` is incorrect, because the body of `A` has not been elaborated, so

Program_Error will be raised on the call to A. In this case GNAT will generate a warning that Program_Error may be raised at the point of the call. Let's look at the warning:

```
 1. procedure x is
 2.    function A return Integer;
 3.    function B return Integer;
 4.    function C return Integer;
 5.
 6.    function B return Integer is begin return A; end;
                                                 |
    >>> warning: call to "A" before body is elaborated may
                 raise Program_Error
    >>> warning: "B" called at line 7
    >>> warning: "C" called at line 9

 7.    function C return Integer is begin return B; end;
 8.
 9.    X : Integer := C;
10.
11.    function A return Integer is begin return 1; end;
12.
13. begin
14.    null;
15. end;
```

Note that the message here says "may raise", instead of the direct case, where the message says "will be raised". That's because whether A is actually called depends in general on run-time flow of control. For example, if the body of B said

```
function B return Integer is
begin
   if some-condition-depending-on-input-data then
      return A;
   else
      return 1;
   end if;
end B;
```

then we could not know until run time whether the incorrect call to A would actually occur, so Program_Error might or might not be raised. It is possible for a compiler to do a better job of analyzing bodies, to determine whether or not Program_Error might be raised, but it certainly couldn't do a perfect job (that would require solving the halting problem and is provably impossible), and because this is a warning anyway, it does not seem worth the effort to do the analysis. Cases in which it would be relevant are rare.

In practice, warnings of either of the forms given above will usually correspond to real errors, and should be examined carefully and eliminated. In the rare case where a warning is bogus, it can be suppressed by any of the following methods:

- Compile with the -gnatws switch set
- Suppress Elaboration_Checks for the called subprogram
- Use pragma Warnings_Off to turn warnings off for the call

For the internal elaboration check case, GNAT by default generates the necessary run-time checks to ensure that Program_Error is raised if any call fails an elaboration check. Of course this can only happen if a warning has been issued as described above. The use of pragma Suppress (Elaboration_Checks) may (but is not guaranteed) to suppress some of these checks, meaning that it may be possible (but is not guaranteed) for a program to be able to call a subprogram whose body is not yet elaborated, without raising a Program_Error exception.

9.5 Controlling Elaboration in GNAT - External Calls

The previous section discussed the case in which the execution of a particular thread of elaboration code occurred entirely within a single unit. This is the easy case to handle, because a programmer has direct and total control over the order of elaboration, and furthermore, checks need only be generated in cases which are rare and which the compiler can easily detect. The situation is more complex when separate compilation is taken into account. Consider the following:

```
package Math is
   function Sqrt (Arg : Float) return Float;
end Math;

package body Math is
   function Sqrt (Arg : Float) return Float is
   begin
      ...
   end Sqrt;
end Math;

with Math;
package Stuff is
   X : Float := Math.Sqrt (0.5);
end Stuff;

with Stuff;
procedure Main is
begin
   ...
end Main;
```

where Main is the main program. When this program is executed, the elaboration code must first be executed, and one of the jobs of the binder is to determine the order in which the units of a program are to be elaborated. In this case we have four units: the spec and body of Math, the spec of Stuff and the body of Main). In what order should the four separate sections of elaboration code be executed?

There are some restrictions in the order of elaboration that the binder can choose. In particular, if unit U has a with for a package X, then you are assured that the spec of X is elaborated before U , but you are not assured that the body of X is elaborated before U. This means that in the above case, the binder is allowed to choose the order:

```
spec of Math
spec of Stuff
body of Math
body of Main
```

but that's not good, because now the call to `Math.Sqrt` that happens during the elaboration of the `Stuff` spec happens before the body of `Math.Sqrt` is elaborated, and hence causes `Program_Error` exception to be raised. At first glance, one might say that the binder is misbehaving, because obviously you want to elaborate the body of something you `with` first, but that is not a general rule that can be followed in all cases. Consider

```
package X is ...

package Y is ...

with X;
package body Y is ...

with Y;
package body X is ...
```

This is a common arrangement, and, apart from the order of elaboration problems that might arise in connection with elaboration code, this works fine. A rule that says that you must first elaborate the body of anything you `with` cannot work in this case (the body of `X` `with`'s `Y`, which means you would have to elaborate the body of `Y` first, but that `with`'s `X`, which means you have to elaborate the body of `X` first, but ... and we have a loop that cannot be broken.

It is true that the binder can in many cases guess an order of elaboration that is unlikely to cause a `Program_Error` exception to be raised, and it tries to do so (in the above example of `Math/Stuff/Spec`, the GNAT binder will in fact always elaborate the body of `Math` right after its spec, so all will be well).

However, a program that blindly relies on the binder to be helpful can get into trouble, as we discussed in the previous sections, so GNAT provides a number of facilities for assisting the programmer in developing programs that are robust with respect to elaboration order.

9.6 Default Behavior in GNAT - Ensuring Safety

The default behavior in GNAT ensures elaboration safety. In its default mode GNAT implements the rule we previously described as the right approach. Let's restate it:

If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a `with`'ed unit, or instantiate a generic unit in a `with`'ed unit, then if the `with`'ed unit does not have pragma `Pure`, `Preelaborate`, or `Elaborate_Body`, then the client should have an `Elaborate_All` for the `with`'ed unit. By following this rule a client is assured that calls and instantiations can be made without risk of an exception.

In this mode GNAT traces all calls that are potentially made from elaboration code, and put in any missing implicit `Elaborate_All` pragmas. The advantage of this approach is that no elaboration problems are possible if the binder can find an elaboration order that is consistent with these implicit `Elaborate_All` pragmas. The disadvantage of this approach is that no such order may exist.

If the binder does not generate any diagnostics, then it means that it has found an elaboration order that is guaranteed to be safe. However, the binder may still be relying on implicitly generated Elaborate_All pragmas so portability to other compilers than GNAT is not guaranteed.

If it is important to guarantee portability, then the compilations should use the -gnatwl (warn on elaboration problems) switch. This will cause warning messages to be generated indicating the missing Elaborate_All pragmas. Consider the following source program:

```
with k;
package j is
  m : integer := k.r;
end;
```

where it is clear that there should be a pragma Elaborate_All for unit k. An implicit pragma will be generated, and it is likely that the binder will be able to honor it. However it is safer to include the pragma explicitly in the source. If this unit is compiled with the -gnatwl switch, then the compiler outputs a warning:

```
1. with k;
2. package j is
3.   m : integer := k.r;
                     |
   >>> warning: call to "r" may raise Program_Error
   >>> warning: missing pragma Elaborate_All for "k"

4. end;
```

and these warnings can be used as a guide for supplying manually the missing pragmas.

9.7 What to do if the Default Elaboration Behavior Fails

If the binder cannot find an acceptable order, it outputs detailed diagnostics. For example:

```
error: elaboration circularity detected
info:   "proc (body)" must be elaborated before "pack (body)"
info:     reason: Elaborate_All probably needed in unit "pack (body)"
info:     recompile "pack (body)" with -gnatwl
info:                             for full details
info:       "proc (body)"
info:         is needed by its spec:
info:       "proc (spec)"
info:         which is withed by:
info:       "pack (body)"
info:  "pack (body)" must be elaborated before "proc (body)"
info:     reason: pragma Elaborate in unit "proc (body)"
```

In this case we have a cycle that the binder cannot break. On the one hand, there is an explicit pragma Elaborate in proc for pack. This means that the body of pack must be elaborated before the body of proc. On the other hand, there is elaboration code in pack that calls a subprogram in proc. This means that for maximum safety, there should really be a pragma Elaborate_All in pack for proc which would require that the body of proc be elaborated before the body of `pack`. Clearly both requirements cannot be satisfied. Faced with a circularity of this kind, you have three different options.

Fix the program

> The most desirable option from the point of view of long-term maintenance is to rearrange the program so that the elaboration problems are avoided. One useful technique is to place the elaboration code into separate child packages. Another is to move some of the initialization code to explicitly called subprograms, where the program controls the order of initialization explicitly. Although this is the most desirable option, it may be impractical and involve too much modification, especially in the case of complex legacy code.

Perform dynamic checks

> If the compilations are done using the `-gnatE` (dynamic elaboration check) switch, then GNAT behaves in a quite different manner. Dynamic checks are generated for all calls that could possibly result in raising an exception. With this switch, the compiler does not generate implicit `Elaborate_All` pragmas. The behavior then is exactly as specified in the Ada 95 Reference Manual. The binder will generate an executable program that may or may not raise Program_Error, and then it is the programmer's job to ensure that it does not raise an exception. Note that it is important to compile all units with the switch, it cannot be used selectively.

Suppress checks

> The drawback of dynamic checks is that they generate a significant overhead at run time, both in space and time. If you are absolutely sure that your program cannot raise any elaboration exceptions, then you can use the `-f` switch for the `gnatbind` step, or `-bargs -f` if you are using `gnatmake`. This switch tells the binder to ignore any implicit `Elaborate_All` pragmas that were generated by the compiler, and suppresses any circularity messages that they cause. The resulting executable will work properly if there are no elaboration problems, but if there are some order of elaboration problems they will not be detected, and unexpected results may occur.

It is hard to generalize on which of these three approaches should be taken. Obviously if it is possible to fix the program so that the default treatment works, this is preferable, but this may not always be practical. It is certainly simple enough to use `-gnatE` or `-f` but the danger in either case is that, even if the GNAT binder finds a correct elaboration order, it may not always do so, and certainly a binder from another Ada compiler might not. A combination of testing and analysis (for which the warnings generated with the `-gnatwl` switch can be useful) must be used to ensure that the program is free of errors. One switch that is useful in this testing is the `-h (horrible elaboration order)` switch for `gnatbind`. Normally the binder tries to find an order that has the best chance of of avoiding elaboration problems. With this switch, the binder plays a devil's advocate role, and tries to choose the order that has the best chance of failing. If your program works even with this switch, then it has a better chance of being error free, but this is still not a guarantee.

For an example of this approach in action, consider the C-tests (executable tests) from the ACVC suite. If these are compiled and run with the default treatment, then all but one of them succeed without generating any error dianostics from the binder. However, there is one test that fails, and this is not surprising, because the whole point of this test is to ensure that the compiler can handle cases where it is impossible to determine a correct order statically, and it checks that an exception is indeed raised at run time.

This one test must be compiled and run using the `-gnatE` switch, and then it passes. Alternatively, the entire suite can be run using this switch. It is never wrong to run with the dynamic elaboration switch if your code is correct, and we assume that the C-tests are indeed correct (it is less efficient, but efficiency is not a factor in running the ACVC tests.)

9.8 Elaboration for Access-to-Subprogram Values

The introduction of access-to-subprogram types in Ada 95 complicates the handling of elaboration. The trouble is that it becomes impossible to tell at compile time which procedure is being called. This means that it is not possible for the binder to analyze the elaboration requirements in this case.

If at the point at which the access value is created, the body of the subprogram is known to have been elaborated, then the access value is safe, and its use does not require a check. This may be achieved by appropriate arrangement of the order of declarations if the subprogram is in the current unit, or, if the subprogram is in another unit, by using pragma `Pure`, `Preelaborate`, or `Elaborate_Body` on the referenced unit.

If the referenced body is not known to have been elaborated at the point the access value is created, then any use of the access value must do a dynamic check, and this dynamic check will fail and raise a `Program_Error` exception if the body has not been elaborated yet. GNAT will generate the necessary checks, and in addition, if the `-gnatwl` switch is set, will generate warnings that such checks are required.

The use of dynamic dispatching for tagged types similarly generates a requirement for dynamic checks, and premature calls to any primitive operation of a tagged type before the body of the operation has been elaborated, will result in the raising of `Program_Error`.

9.9 Summary of Procedures for Elaboration Control

First, compile your program with the default options, using none of the special elaboration control switches. If the binder successfully binds your program, then you can be confident that, apart from issues raised by the use of access-to-subprogram types and dynamic dispatching, the program is free of elaboration errors. If it is important that the program be portable, then use the `-gnatwl` switch to generate warnings about missing `Elaborate_All` pragmas, and supply the missing pragmas.

If the program fails to bind using the default static elaboration handling, then you can fix the program to eliminate the binder message, or recompile the entire program with the `-gnatE` switch to generate dynamic elaboration checks, or, if you are sure there really are no elaboration problems, use the `-f` switch for the binder to cause it to ignore implicit `Elaborate_All` pragmas generated by the compiler.

10 The cross-referencing tools gnatxref and gnatfind

The compiler generates cross-referencing information (unless you set the '-gnatx' switch), which are saved in the '.ali' files. This information indicates where in the source each entity is declared and referenced.

The two tools gnatxref and gnatfind take advantage of this information to provide the user with the capability to easily locate the declaration and references to an entity. These tools are quite similar, the difference being that gnatfind is intended for locating definitions and/or references to a specified entity or entities, whereas gnatxref is oriented to generating a full report of all cross-references.

To use these tools, you must not compile your application using the '-gnatx' switch on the 'gnatmake' command line (See Info file 'gnat_ug', node 'The GNAT Make Program gnatmake'). Otherwise, cross-referencing information will not be generated.

10.1 Common switches

The following switches are recognized by both gnatxref and gnatfind. They are used to specify the configuration of the tools, such as the default search paths for source and object files.

'-a' If this switch is present, gnatfind and gnatxref will parse the read-only files found in the library search path. Otherwise, these files will be ignored. This option can be used to protect Gnat sources or your own libraries from beeing parsed, thus making gnatfind and gnatxref much faster, and their output much smaller.

'-aIDIR' When looking for source files also look in directory DIR. The order in which source file search is undertaken is the same as for 'gnatmake'.

'-aODIR' When searching for library and object files, look in directory DIR. The order in which library files are searched is the same as for 'gnatmake'.

'-f' If this switch is set, the output file names will be preceded by their directory (if the file was found in the search path). If this switch is not set, the directory will not be printed.

'-g' If this switch is set, information is output only for library-level entities, ignoring local entities. The use of this switch may accelerate gnatfind and gnatxref.

'-IDIR' Equivalent to '-aODIR -aIDIR'.

'-pFILE' Specify a project file (See Section 10.4 [Project files], page 73) to use. By default, gnatxref and gnatfind will try to locate a project file in the current directory.

If a project file is either specified or found by the tools, then the content of the source directory and object directory lines are added as if they had been specified respectively by '-aI' and '-aO'.

All these switches (plus the others found in the next two pages) may be in any order on the command line, and may even appear after the file names. They need not be separated by spaces, thus you can say 'gnatfind -ag' instead of 'gnatfind -a -g'.

10.2 Gnatxref switches

The command lines for `gnatxref` is:

```
gnatxref [switches] sourcefile1 [sourcefile2 ...]
```

where

'`sourcefile1, sourcefile2`'
identifies the source files for which a report is to be generated. The 'with'ed units will be processed too. You must provide at least one file.

The switches can be (including the global switches defined previously) :

'`-u`' Output only unused symbols. This may be really useful if you give your main compilation unit on the command line, as `gnatxref` will then display every unused entity and 'with'ed package.

'`-v`' Instead of producing the default output, `gnatxref` will generate a '`tags`' file that can be used by vi. See Section 10.6 [Examples of gnatxref usage], page 75 for examples how to use this feature. The tags file is output to the standard output, thus you will have to redirect it to a file.

10.3 gnatfind switches

The command line for `gnatfind` is:

```
    gnatfind [switches] pattern[:sourcefile[:line[:column]]] [file1 file2
```

where

'`pattern`' An entity will be output only if it matches the regular expression found in '`pattern`' (See Section 10.5 [Regular expressions in gnatfind and gnatxref], page 74).

Omitting the pattern is equivalent to specifying '`*`', which will match any entity. Note that if you do not provide a pattern, you have to provide both a sourcefile and a line.

Entity names are given in Latin-1, with upper-lower case equivalence for matching purposes. At the current time there is no support for 8-bit codes other than Latin-1, or for wide characters in identifiers.

'`sourcefile`'
`gnatfind` will look for references, bodies or declarations of symbols referenced in '`sourcefile`', at line '`line`' and column '`column`'. see see Section 10.7 [Examples of gnatfind usage], page 76 for syntax examples.

'`line`' is a decimal integer identifying the line number containing the reference to the entity (or entities) to be located.

'`column`' is a decimal integer identifying the exact location on the line of the first character of the identifier for the entity reference. Columns are numbered from 1.

'`file1 file2 ...`'
: The search will be restricted to these files. If none are given, then the search will be done for every library file in the search path. These file must appear only after the pattern or sourcefile.

At least one of 'sourcefile' or 'pattern' has to be present on the command line.

Two switches are available:

'`-e`' By default, `gnatfind` accept the simple regular expression set for '`pattern`'. If this switch is set, then the pattern will be considered as full Unix-style regular expression.

'`-r`' By default, `gnatfind` will output only the information about the declaration, body or type completion of the entities. If this switch is set, the `gnatfind` will locate every reference to the entities in the files specified on the command line (or in every file in the search path if no file is given on the command line).

10.4 Project files

The project files allows a programmer to specify how to compile its application, where to find sources,... These files are used primarily by the Emacs Ada mode, but they can also be used by the two tools `gnatxref` and `gnatfind`.

A project file name must end with '`.adp`'. If a single one is present in the current directory, then `gnatxref` and `gnatfind` will extract the information from it. If multiple project files are found, none of them is read, and you have to use the '`-p`' switch to specify the one you want to use.

The following lines can be included, even though most of them have default values which can be used in most cases. The lines can be entered in any order in the file.

'`src_dir=DIR [default: "./"]`'
: specifies a directory where to look for source files. Multiple src_dir lines can be specified and they will be searched in the order they are specified.

'`obj_dir=DIR [default: "./"]`'
: specifies a directory where to look for object and library files. Multiple obj_dir lines can be specified and they will be searched in the order they are specified

'`comp_opt=SWITCHES [default: ""]`'
: creates a variable which can be referred to subsequently by using the '`${comp_opt}`' notation. This is intended to store the default switches given to '`gnatmake`' and '`gcc`'.

'`bind_opt=SWITCHES [default: ""]`'
: creates a variable which can be referred to subsequently by using the '`${bind_opt}`' notation. This is intended to store the default switches given to '`gnatbind`'.

'`link_opt=SWITCHES [default: ""]`'
: creates a variable which can be referred to subsequently by using the '`${link_opt}`' notation. This is intended to store the default switches given to '`gnatlink`'.

‘`main=EXECUTABLE [default: ""]`’

specifies the name of the executable for the application. This variable can be referred to in the following lines bu using the ‘`${main}`’ notation.

‘`comp_cmd=COMMAND [default: "gcc -I${src_dir} -c -g -gnatq"]`’

specified the command used to compile a single file in the application. This line must include the ‘`-gnatx`’ switch, as its main usage is to recompile a file to update the cross-referencing information.

‘`make_cmd=COMMAND [default: "gnatmake ${main} -aI${src_dir}`’

-aO${obj_dir} -g -gnatq -cargs ${comp_opt} -bargs ${bind_opt} -largs ${link_opt}"]

specifies the command used to recompile the whole application. It too must include the ‘`-gnatx`’ switch.

‘`run_cmd=COMMAND [default: "${main}"]`’

specifies the command used to run the application.

‘`debug_cmd=COMMAND [default: "gdb ${main}"]`’

specifies the command used to debug the application

`gnatxref` and `gnatfind` only take into account the ‘`src_dir`’ and ‘`obj_dir`’ lines, and ignore the others.

10.5 Regular expressions in gnatfind and gnatxref

As specified in the section about `gnatfind`, the pattern can be a regular expression. Actually, there are to set of regular expressions which are recognized by the program :

‘`globbing patterns`’

These are the most usual regular expression. They are the same that you generally used in a Unix shell command line, or in a DOS session.

Here is a more formal grammar :

```
regexp ::= term
term   ::= elmt            -- matches elmt
term   ::= elmt elmt       -- concatenation (elmt then elmt)
term   ::= *               -- any string of 0 or more character
term   ::= ?               -- matches any character
term   ::= [char {char}] -- matches any character listed
term   ::= [char - char]   -- matches any character in range
```

‘`full regular expression`’

The second set of regular expressions is much more powerful. This is the type of regular expressions recognized by utilities such a ‘`grep`’.

The following is the form of a regular expression, expressed in Ada reference manual style BNF is as follows

```
regexp ::= term {| term} -- alternation (term or term ...)

term ::= item {item}     -- concatenation (item then item)

item ::= elmt              -- match elmt
item ::= elmt *            -- zero or more elmt's
item ::= elmt +            -- one or more elmt's
item ::= elmt ?            -- matches elmt or nothing

elmt ::= nschar            -- matches given character
elmt ::= [nschar {nschar}]   -- matches any character listed
elmt ::= [^ nschar {nschar}] -- matches any character not list
elmt ::= [char - char]     -- matches chars in given range
elmt ::= \ char            -- matches given character
elmt ::= .                 -- matches any single character
elmt ::= ( regexp )        -- parens used for grouping

char ::= any character, including special characters
nschar ::= any character except ()[].*+?^
```

Following are a few examples :

'`abcde|fghi`'
: will match any of the two strings 'abcde' and 'fghi'.

'`abc*d`' will match any string like 'abd', 'abcd', 'abccd', 'abcccd', and so on

'`[a-z]+`' will match any string which has only lower-case characters in it (and at least one character

10.6 Examples of gnatxref usage

10.6.1 General usage

For the following examples, we will consider the following units :

```
main.ads:
1: with Bar;
2: package Main is
3:     procedure Foo (B : in Integer);
4:     C : Integer;
5: private
6:     D : Integer;
7: end Main;

main.adb:
1: package body Main is
2:     procedure Foo (B : in Integer) is
3:     begin
4:        C := B;
5:        D := B;
6:        Bar.Print (B);
7:        Bar.Print (C);
8:     end Foo;
```

```
9: end Main;

bar.ads:
1: package Bar is
2:     procedure Print (B);
3: end bar;
```

`gnatxref main.adb`

`gnatxref` generates cross-reference information for main.adb and every unit 'with'ed by main.adb.

The output would be:

```
## Entity_Name Type Declaration [Body] {Modifications} {Referenc
Main procedure main.ads:2:9 main.adb:1:14 {} {}
Foo procedure main.ads:3:15 main.adb:2:15 {} {}
C Integer main.ads:4:5 {main.adb:4:8} {main.adb:7:19}
Bar procedure bar.ads:1:9 {} {main.ads:1:6}
Print procedure bar.ads:2:15 bar.adb:3:10 {} {main.adb:6:12 7:12
```

that is the entity `Main` is declared in main.ads, line 2, column 9, its body is in main.adb, line 1, column 14 and is not referenced any where.

The entity `Print` is declared in bar.ads, line 2, column 15, its body is in bar.adb, line 3, column 10, and it it referenced in main.adb, line 6 column 12 and line 7 column 12.

`gnatxref package1.adb package2.ads`

`gnatxref` will generates cross-reference information for package1.adb, package2.ads and any other package 'with'ed by any of these.

10.6.2 Using gnatxref with vi

`gnatxref` can generate a tags file output, which can be used directly from '`vi`'. Note that the standard version of '`vi`' will not work properly with overloaded symbols. Consider using another free implementation of '`vi`', such as '`vim`'.

```
gnatxref -v gnatfind.adb > tags
```

will generate the tags file for `gnatfind` itself (if the sources are in the search path!).

From '`vi`', you can then use the command '`:tag <entity>`' (replacing `<entity>` by whatever you are looking for), and vi will display a new file with the corresponding declaration of entity.

10.7 Examples of `gnatfind` usage

`gnatfind -f xyz:main.adb`

Find declarations for all entities xyz referenced at least once in main.adb. The references are search in every library file in the search path.

The directories will be printed as well (as the '`-f`' switch is set)

The output will look like:

```
          directory/main.ads:106:14 xyz  <= declaration
          directory/main.adb:24:10 xyz <= body
          directory/foo.ads:45:23 <= declaration
```

that is to say, one of the entities xyz found in main.adb is declared at line 12 of main.ads (and its body is in main.adb), and another one is declared at line 45 of foo.ads

`gnatfind -r "*x*":main.ads:123 foo.adb`

Find references to all entities containing an x that are referenced on line 123 of main.ads. The references will be searched only in main.adb and foo.adb.

`gnatfind main.ads:123`

Find declarations and bodies for all entities that are referenced on line 123 of main.ads.

This is the same as `gnatfind "*":main.adb:123`.

`gnatfind mydir/main.adb:123:45`

Find the declaration for the entity referenced at column 45 in line 123 of file main.adb in directory mydir. Note that it is usual to omit the identifier name when the column is given, since the column position identifies a unique reference.

The column has to be the beginning of the identifier, and should not point to any character in the middle of the identifier.

11 File Name Krunching Using gnatkr

This chapter discusses the method used by the compiler to shorten the default file names chosen for Ada units so that they do not exceed the maximum length permitted. It also describes the `gnatkr` utility that can be used to determine the result of applying this shortening.

11.1 About gnatkr

The default file naming rule in GNAT is that the file name must be derived from the unit name. The exact default rule is as follows:

- Take the unit name and replace all dots by hyphens.
- If such a replacement occurs in the second character position of a name, and the first character is a, g, s, or i then replace the dot by the character ~ (tilde) instead of a minus.

The reason for this exception is to avoid clashes with the standard names for children of System, Ada, Interfaces, and GNAT, which use the prefixes s- a- i- and g- respectively.

The `-gnatk`*nn* switch of the compiler activates a "krunching" circuit that limits file names to nn characters (where nn is a decimal integer). For example, using OpenVMS, where the maximum file name length is 39, the value of nn is usually set to 39, but if you want to generate a set of files that would be usable if ported to a system with some different maximum file length, then a different value can be specified. The default value of 39 for OpenVMS need not be specified.

The `gnatkr` utility can be used to determine the krunched name for a given file, when krunched to a specified maximum length.

11.2 Using gnatkr

The `gnatkr` command has the form

```
$ gnatkr name [length]
```

name can be an Ada name with dots or the GNAT name of the unit, where the dots representing child units or subunit are replaced by hyphens. The only confusion arises if a name ends in `.ads` or `.adb`. `gnatkr` takes this to be an extension if there are no other dots in the name and the whole name is in lowercase.

length represents the length of the krunched name. The default when no argument is given is 8 characters. A length of zero stands for unlimited, in other words do not chop except for system files which are always 8.

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

11.3 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 specification 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  14
  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:

  '`ada-`' replaced by '`a-`'

  '`gnat-`' replaced by '`g-`'

  '`interfaces-`'
  replaced by '`i-`'

  '`system-`' replaced by '`s-`'

  These system files have a hyphen in the second character position. That is why normal user files replace such a character with a tilde, to avoid confusion with system file names.

  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, and if file name krunching is used then it is your responsibility to ensure that no name clashes occur. The utility program **gnatkr** is supplied for conveniently determining the krunched name of a file.

11.4 Examples of gnatkr Usage

```
$ gnatkr very_long_unit_name.ads      --> velounna.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
```

12 Preprocessing Using gnatprep

The gnatprep utility provides a simple preprocessing capability for Ada programs. It is designed for use with GNAT, but is not dependent on any special features of GNAT.

12.1 Using gnatprep

To call gnatprep use

```
$ gnatprep infile outfile deffile switches
```

where

`infile` is the full name of the input file, which is an Ada source file containing preprocessor directives.

`outfile` is the full name of the output file, which is an Ada source in standard Ada form. When used with GNAT, this file name will normally have an ads or adb suffix.

`deffile` is the full name of a text file containing definitions of symbols to be referenced by the preprocessor.

`switches` is an optional sequence of switches as described in the next section.

12.2 Switches for gnatprep

`-c` Causes both preprocessor lines and the lines deleted by preprocessing to be retained in the output source as comments marked with the special string "–! ". This option will result in line numbers being preserved in the output file.

`-b` Causes both preprocessor lines and the lines deleted by preprocessing to be replaced by blank lines in the output source file, preserving line numbers in the output file.

`-r` Causes a `Source_Reference` pragma to be generated that references the original input file, so that error messages will use the file name of this original file. The use of this switch implies that preprocessor lines are not to be removed from the file, so its use will force `-b` mode if `-c` has not been specified explicitly.

Note that if the file to be preprocessed contains multiple units, then it will be necessary to `gnatchop` the output file from `gnatprep`. If a `Source_Reference` pragma is present in the preprocessed file, it will be respected by `gnatchop -r` so that the final chopped files will correctly refer to the original input source file for `gnatprep`.

`-s` Causes a sorted list of symbol names and values to be listed on the standard output file.

`-u` Causes undefined symbols to be treated as having the value FALSE in the context of a preprocessor test. In the absence of this option, an undefined symbol in a `#if` or `#elsif` test will be treated as an error.

`-v` Causes gnatprep to output a copyright notice including the version number of gnatprep.

Note: if neither `-b` nor `-c` is present, then preprocessor lines and deleted lines are completely removed from the output, unless -r is specified, in which case -b is assumed.

12.3 Form of definitions file

The definitions file contains lines of the form

```
symbol := value
```

where symbol is an identifier, following normal Ada (case-insensitive) rules for its syntax, and value is one of the following:

- Empty, corresponding to a null substitution
- A string literal using normal Ada syntax
- Any sequence of characters from the set (letters, digits, period, underline).

Comment lines may also appear in the definitions file, starting with the usual `--`, and comments may be added to the definitions lines.

12.4 Form of input text for gnatprep

The input text may contain preprocessor conditional inclusion lines, as well as general symbol substitution sequences. The preprocessor conditional inclusion commands have the form

```
#if [not] symbol [then]
   lines
#elsif [not] symbol [then]
   lines
#elsif [not] symbol [then]
   lines
...
#else
   lines
#end if;
```

For these Boolean tests, the symbol must have either the value True or False, that is to say the right-hand of the symbol definition must be one of the (case-insensitive) literals True or False . If the value is True, then the corresponding lines are included, and if the value is False, they are excluded.

If the symbol referenced is not defined in the symbol definitions file, then the effect depends on whether or not switch `-u` is specified. If so, then the symbol is treated as if it had the value `False` and the test fails. If this switch is not specified, then it is an error to reference an undefined symbol. It is also an error to reference a symbol that is defined with a value other than `True` or `False`.

The use of the not operator inverts the sense of this logical test, so that the lines are included only if the symbol is not defined. The `then` keyword is optional as shown

The # must be in column one, but otherwise the format is free form. Spaces or tabs may appear between the # and the keyword. The keywords and the symbols are case insensitive as in normal Ada code. Comments may be used on a preprocessor line, but other than that, no other tokens may appear on a preprocessor line. Any number of `elsif` clauses can be present, including none at all. The `else` is optional, as in Ada.

The # marking the start of a preprocessor line must be the first non-blank character on the line, i.e. it must be preceded only by spaces or horizontal tabs.

Symbol substitution outside of preprocessor lines is obtained by using the sequence

```
$symbol
```

anywhere within a source line, except in a comment. The identifier following the $ must match one of the symbols defined in the symbol definition file, and the result is to substitute the value of the symbol in place of `$symbol` in the output file.

13 The GNAT run-time library builder `gnatlbr`

`gnatlbr` is a tool for rebuilding the GNAT run-time with user supplied configuration pragmas.

13.1 Running `gnatlbr`

The `gnatlbr` command has the form

```
$ gnatlbr --[create | set | delete]=directory --config=file
```

13.2 Switches for `gnatlbr`

`gnatlbr` recognizes the following switches:

`--create=directory`
Create the new run-time library in the specified directory.

`--set=directory`
Make the library in the specified directory the current run-time library.

`--delete=directory`
Delete the run-time library in the specified directory.

`--config=file`
With –create: Use the configuration pragmas in the specified file when building the library.

With –set: Use the configuration pragmas in the specified file when compiling.

13.3 Example of `gnatlbr` Usage

14 The GNAT library browser gnatls

`gnatls` is a tool that outputs information about compiled units. It gives the relationship between objects, unit names and source files. It can also be used to check the source dependencies of a unit as well as various characteristics.

14.1 Running gnatls

The `gnatls` command has the form

```
$ gnatls switches object_or_ali_file
```

The main argument is the list of object or '`ali`' files (see Section 2.7 [The Ada Library Information Files], page 13) for which information is requested.

In normal mode, without additional option, `gnatls` produces a four-column listing. Each line represents information for a specific object. The first column gives the full path of the object, the second column gives the name of the principal unit in this object, the third column gives the status of the source and the fourth column gives the full path of the source representing this unit. Here is a simple example of use:

```
$ gnatls *.o
./demo1.o            demo1            DIF demo1.adb
./demo2.o            demo2             OK demo2.adb
./hello.o            h1                OK hello.adb
./instr-child.o      instr.child      MOK instr-child.adb
./instr.o            instr             OK instr.adb
./tef.o              tef              DIF tef.adb
./text_io_example.o  text_io_example   OK text_io_example.adb
./tgef.o             tgef             DIF tgef.adb
```

The first line can be interpreted as follows: the main unit which is contained in object file '`demo1.o`' is demo1, whose main source is in '`demo1.adb`'. Furthermore, the version of the source used for the compilation of demo1 has been modified (DIF). Each source file has a status qualifier which can be:

`OK (unchanged)`
: The version of the source file used for the compilation of the specified unit corresponds exactly to the actual source file.

`MOK (slightly modified)`
: The version of the source file used for the compilation of the specified unit differs from the actual source file but not enough to require recompilation. If you use gnatmake with the qualifier `-m (minimal recompilation)`, a file marked MOK will not be recompiled.

`DIF (modified)`
: No version of the source found on the path corresponds to the source used to build this object.

`??? (file not found)`
: No source file was found for this unit.

`HID (hidden, unchanged version not first on PATH)`
: The version of the source that corresponds exactly to the source used for compilation has been found on the path but it is hidden by another version of the same source that has been modified.

14.2 Switches for `gnatls`

`gnatls` recognizes the following switches:

`-a` Consider all units, including those of the predefined Ada library. Especially useful with `-d`.

`-d` List sources from which specified units depend on.

`-h` Output the list of options.

`-o` Only output information about object files.

`-s` Only output information about source files.

`-u` Only output information about compilation units.

`-aO`*dir*
`-aI`*dir*
`-I`*dir*

`-I-` Source and Object path manipulation. Same meaning as the equivalent $ gnatmake flags Section 6.2 [Switches for gnatmake], page 46

`-v` Verbose mode. Output the complete source and object paths. Do not use the default column layout but instead use long format giving as much as information possible on each requested units, including special characteristics such as:

`Preelaborable`
: The unit is preelaborable in the Ada 95 sense.

`No_Elab_Code`
: No elaboration code has been produced by the compiler for this unit.

`Pure` The unit is pure in the Ada 95 sense.

`Elaborate_Body`
: The unit contains a pragma Elaborate_Body.

`Remote_Types`
: The unit contains a pragma Remote_Types.

`Shared_Passive`
: The unit contains a pragma Shared_Passive.

`Predefined`
: This unit is part of the predefined environment and cannot be modified by the user.

`Remote_Call_Interface`
: The unit contains a pragma Remote_Call_Interface.

14.3 Example of gnatls Usage

Example of using the verbose switch. Note how the source and object paths are affected by the -I switch.

```
$ gnatls -v -I.. demo1.o

GNATLS 3.10w (970212) Copyright 1997 Free Software Foundation, Inc.

Source Search Path:
   <Current_Directory>
   ../
   /home/comar/local/adainclude/


Object Search Path:
   <Current_Directory>
   ../
   /home/comar/local/lib/gcc-lib/mips-sni-sysv4/2.7.2/adalib/

./demo1.o
   Unit =>
     Name   => demo1
     Kind   => subprogram body
     Flags  => No_Elab_Code
     Source => demo1.adb    modified
```

The following is an example of use of the dependency list. Note the use of the -s switch which gives a straight list of source files. This can be useful for building specialized scripts.

```
$ gnatls -d demo2.o
./demo2.o   demo2        OK demo2.adb
                         OK gen_list.ads
                         OK gen_list.adb
                         OK instr.ads
                         OK instr-child.ads

$ gnatls -d -s -a demo1.o
demo1.adb
/home/comar/local/adainclude/ada.ads
/home/comar/local/adainclude/a-finali.ads
/home/comar/local/adainclude/a-filico.ads
/home/comar/local/adainclude/a-stream.ads
/home/comar/local/adainclude/a-tags.ads
gen_list.ads
gen_list.adb
/home/comar/local/adainclude/gnat.ads
/home/comar/local/adainclude/g-io.ads
instr.ads
/home/comar/local/adainclude/system.ads
/home/comar/local/adainclude/s-exctab.ads
```

```
/home/comar/local/adainclude/s-finimp.ads
/home/comar/local/adainclude/s-finroo.ads
/home/comar/local/adainclude/s-secsta.ads
/home/comar/local/adainclude/s-stalib.ads
/home/comar/local/adainclude/s-stoele.ads
/home/comar/local/adainclude/s-stratt.ads
/home/comar/local/adainclude/s-tasoli.ads
/home/comar/local/adainclude/s-unstyp.ads
/home/comar/local/adainclude/unchconv.ads
```

15 Finding memory problems with gnatmem

gnatmem, is a tool that monitors dynamic allocation and deallocation activity in a program, and displays information about incorrect deallocations and possible sources of memory leaks. Gnatmem provides three type of information:

- General information concerning memory management, such as the total number of allocations and deallocations, the amount of allocated memory and the high water mark, i.e. the largest amount of allocated memory in the course of program execution.
- Backtraces for all incorrect deallocations, that is to say deallocations which do not correspond to a valid allocation.
- Information on each allocation that is potentially the origin of a memory leak.

15.1 Running gnatmem

The gnatmem command has the form

```
    $ gnatmem [n] [-o file] user_program [program_arg]*
  or
    $ gnatmem [n] -i file
```

Gnatmem must be supplied with the executable to examine, followed by its run-time inputs. For example, if a program is executed with the command:

```
    $ my_program arg1 arg2
```

then it can be run under gnatmem control using the command:

```
    $ gnatmem my_program arg1 arg2
```

The program is transparently executed under the control of the debugger Section 20.1 [The GNAT Debugger GDB], page 111. This does not affect the behavior of the program, except for sensitive real-time programs. When the program has completed execution, gnatmem outputs a report containing general allocation/deallocation information and potential memory leak. For better results, the user program should be compiled with debugging options Section 3.2 [Switches for gcc], page 22.

Here is a simple example of use:

*************** debut cc

```
$ gnatmem test_gm

Global information
------------------
   Total number of allocations        :  45
   Total number of deallocations      :   6
   Final Water Mark (non freed mem)   :  11.29 Kilobytes
   High Water Mark                    :  11.40 Kilobytes

.
.
.
Allocation Root # 2
```

```
-------------------
 Number of non freed allocations    :  11
 Final Water Mark (non freed mem)   :   1.16 Kilobytes
 High Water Mark                    :   1.27 Kilobytes
 Backtrace                          :
   test_gm.adb:23 test_gm.alloc
.
.
.
```

The first block of output give general information. In this case, the Ada construct "new" was executed 45 times, and only 6 calls to an unchecked deallocation routine occurred.

Subsequent paragraphs display information on all allocation roots. An allocation root is a specific point in the execution of the program that generates some dynamic allocation, such as a "new" construct. This root is represented by an execution backtrace (or subprogram call stack). By default the backtrace depth for allocations roots is 1, so that a root corresponds exactly to a source location. The backtrace can be made deeper, to make the root more specific.

15.2 Switches for gnatmem

`gnatmem` recognizes the following switches:

`n` N is an integer literal (usually between 1 and 10) which controls the depth of the backtraces defining allocation root. The default value for N is 1. The deeper the backtrace, the more precise the localization of the root. Note that the total number of roots can depend on this parameter.

`-o file` Direct the gdb output to the specified file. The gdb script used to generate this output is also saved in the file '`gnatmem.tmp`'.

`-i file` Do the `gnatmem` processing starting from `file` which has been generated by a previous call to `gnatmem` with the -o switch. This is useful for post mortem processing.

15.3 Example of gnatmem Usage

The first example shows the use of `gnatmem` on a simple leaking program. Suppose that we have the following Ada program:

```
with Unchecked_Deallocation;
procedure Test_Gm is

   type T is array (1..1000) of Integer;
   type Ptr is access T;
   procedure Free is new Unchecked_Deallocation (T, Ptr);
   A : Ptr;

   procedure My_Alloc is
   begin
```

```
         A := new T;
      end My_Alloc;

      procedure My_DeAlloc is
         B : Ptr := A;
      begin
         Free (B);
       end My_DeAlloc;

   begin
      My_Alloc;
      for I in 1 .. 5 loop
         for J in I .. 5 loop
            My_Alloc;
         end loop;
         My_Dealloc;
      end loop;
   end;
```

The program needs to be compiled with debugging option:

```
$ gnatmake -g test_gm
```

gnatmem is invoked simply with

```
$ gnatmem test_gm
```

which produces the following output:

```
Global information
------------------
   Total number of allocations        :  18
   Total number of deallocations      :   5
   Final Water Mark (non freed mem)   :  53.00 Kilobytes
   High Water Mark                    :  56.90 Kilobytes

Allocation Root # 1
-------------------
 Number of non freed allocations    :  11
 Final Water Mark (non freed mem)   :  42.97 Kilobytes
 High Water Mark                    :  46.88 Kilobytes
 Backtrace                          :
   test_gm.adb:11 test_gm.my_alloc

Allocation Root # 2
-------------------
 Number of non freed allocations    :   1
 Final Water Mark (non freed mem)   :  10.02 Kilobytes
 High Water Mark                    :  10.02 Kilobytes
 Backtrace                          :
   s-secsta.adb:81 system.secondary_stack.ss_init
```

```
Allocation Root # 3
-------------------
 Number of non freed allocations    :   1
 Final Water Mark (non freed mem)   :  12 Bytes
 High Water Mark                    :  12 Bytes
 Backtrace                          :
   s-secsta.adb:181 system.secondary_stack.ss_init
```

Note that the GNAT run time contains itself a certain number of allocations that havk no corresponding deallocation, as shown here for root #2 and root #1. This is a normal behavior when the number of non freed allocations is one, it locates dynamic data structures that the run time needs for the complete lifetime of the program. Note also that there is only one allocation root in the user program with a single line back trace: test_gm.adb:11 test_gm.my_alloc, whereas a careful analysis of the program shows that 'My_Alloc' is called at 2 different points in the source (line 21 and line 24). If those two allocation roots need to be distinguished, the backtrace depth parameter can be used:

```
$ gnatmem 3 test_gm
```

which will give the following output:

```
Global information
------------------
   Total number of allocations        :  18
   Total number of deallocations      :   5
   Final Water Mark (non freed mem)   :  53.00 Kilobytes
   High Water Mark                    :  56.90 Kilobytes

Allocation Root # 1
-------------------
 Number of non freed allocations    :  10
 Final Water Mark (non freed mem)   :  39.06 Kilobytes
 High Water Mark                    :  42.97 Kilobytes
 Backtrace                          :
   test_gm.adb:11 test_gm.my_alloc
   test_gm.adb:24 test_gm
   b_test_gm.c:52 main

Allocation Root # 2
-------------------
 Number of non freed allocations    :   1
 Final Water Mark (non freed mem)   :  10.02 Kilobytes
 High Water Mark                    :  10.02 Kilobytes
 Backtrace                          :
   s-secsta.adb:81  system.secondary_stack.ss_init
   s-secsta.adb:283 <system__secondary_stack___elabb>
   b_test_gm.c:33   adainit

Allocation Root # 3
-------------------
 Number of non freed allocations    :   1
```

```
     Final Water Mark (non freed mem)   :   3.91 Kilobytes
     High Water Mark                    :   3.91 Kilobytes
     Backtrace                          :
       test_gm.adb:11 test_gm.my_alloc
       test_gm.adb:21 test_gm
       b_test_gm.c:52 main

    Allocation Root # 4
    -------------------
     Number of non freed allocations    :   1
     Final Water Mark (non freed mem)   :  12 Bytes
     High Water Mark                    :  12 Bytes
     Backtrace                          :
       s-secsta.adb:181 system.secondary_stack.ss_init
       s-secsta.adb:283 <system__secondary_stack___elabb>
       b_test_gm.c:33   adainit
```

The allocation root #1 of the first example has been split in 2 roots #1 and #3 thanks to the more precise associated backtrace.

15.4 Implementation note

`gnatmem` executes the user program under the control of `gdb` using a script that sets breakpoints and gathers information on each dynamic allocation and deallocation. The output of the script is then analyzed by `gnatmem` in order to locate memory leaks and their origin in the program. Gnatmem works by recording each address returned by the allocation procedure (`__gnat_malloc`) along with the backtrace at the allocation point. On each deallocation, the deallocated address is matched with the corresponding allocation. At the end of the processing, the unmatched allocations are considered potential leaks. All the allocations associated with the same backtrace are grouped together and form an allocation root. The allocation roots are then sorted so that those with the biggest number of unmatched allocation are printed first. A delicate aspect of this technique is to distinguish betwen the data produced by the user program and the data produced by the gdb script. Currently, on systems that allow probing the terminal, the gdb command `"tty"` is used to force the program output to be redirected to the current terminal while the gdb output is directed to a file or to a pipe in order to be processed subsequently by `gnatmem`.

16 ASIS-Based Tools

Some of the tools distributed with GNAT are based on the ASIS implementation for GNAT (ASIS-for-GNAT). Binary executables for such tools do not require ASIS-for-GNAT to be around and they have a command-line interface similar to other GNAT tools. The main specific feature of ASIS-based tools is that they process tree output files.

16.1 The ASIS Implementation for GNAT (ASIS-for-GNAT)

The ASIS implementation for GNAT, called ASIS-for-GNAT, is the implementation if the Ada Semantic Interface Specification (ASIS). It is a separate product which is not included in the standard GNAT distribution. However, the binary executables for tools created on top of ASIS-for-GNAT do not require ASIS-for-GNAT installed on your system and they can be used for a standard GNAT distribution along with other GNAT tools

16.2 Tree Files

The ASIS implementation for GNAT is based on tree output files (or, simply, tree files). A tree file stores a snapshot of the compiler internal data structures in the very end of a successful compilation. It contains all the syntactical and semantic information about the unit being compiled and all the units upon which it depends semantically. ASIS-for-GNAT (and, therefore, any tool based on its top) processes tree files, extracts this information from it and converts it into the format prescribing by the ASIS definition.

To use some ASIS-based tools, a user should take care of producing the right set of tree files for the tool, some other ASIS tools produce a needed set of tree files themselves.

GNAT produces a tree file if -gnatt option is set when calling gcc. ASIS needs tree files created in "compile-only" GNAT mode set by -gnatc gcc switch. Names of the tree files are obtained by replacing 'b' with 't' in the extension of the name of the source file being compiled.

Therefore, to produce a tree file for the body of a procedure Foo contained in the source file named 'foo.adb', you should compile it as

```
$ gcc -c -gnatc -gnatt foo.adb
```

and you should get the tree file named 'foo.atb' as a result of such a compilation.

17 Creating Sample Bodies Using gnatstub

gnatstub creates body samples - that is, empty but compilable bodies for library unit declarations.

gnatstub is an ASIS-based tool, but it creates a needed tree file itself, so it can be considered as a usual command-line utility program when using with GNAT.

To create a body sampler, gnatstub has to compile the library unit declaration. Therefore, bodies can be created only for legal library units. Moreover, if a library unit depends semantically upon units located not only in the current directory, you have to provide a source search path when calling gnatstub, see the description of gnatstub switches below.

17.1 Running gnatstub

gnatstub has the command-line interface of the form

```
$ gnatstub [switches] filename [directory]
```

where

`filename` is the name of a source file containing a library unit declaration to create a body for. This name should follow the GNAT file name conventions. No crunching is allowed for this file name. The file name may contain the path information.

`directory`
indicates the directory to place a sample body (default is the current directory)

`switches` is an optional sequence of switches as described in the next section

17.2 Switches for gnatstub

`-f` Replace an existing body file (if any) with a body sample. If the destination directory already contains a file which name has a form of the body file for the argument spec file, gnatstub replaces it with the body sample if `-f` switch is set or leaves it intact otherwise.

`-hs` Put in body sample the comment header from the source of the library unit declaration ("comment header" is all the comments preceding the compilation unit).

`-hg` Put in body sample a sample comment header

`-Idir`
`-I-` These switches have just the same meaning as in calls to gcc or gnatmake. They are used to define the source search path in the call to gcc issued by gnatstub to compile an argument source file to create a tree file.

`-i`*n* (*n* is a decimal integer in the range of 1 to 9) Sets the indentation level in the generated body sample to n.

`-k` Do not remove the tree file: as default, after creating the body sampler gnatstub removes from the current directory the tree file created for the argument source file. `-k` prevents deleting the tree file.

`-l`*n* (*n* is a decimal integer in the range of 60 to 999) Sets maximum line length in a body sample to n.

`-q` Quiet mode: gnatstub does not generate a confirmation when a body is successfully created or a message when a body is not required for an argument unit.

`-r` Reuse the tree file (if any) instead of creating it: instead of creating the tree file for the library unit declaration, gnatstub tries to find it in the current directory and to use it for creating a body. If the tree file is not found, no body is created. `-r` also implies `-k`, whether or not `-k` is set explicitly.

`-t` Overwrite the existing tree file: if the current directory already contains the file which, according to the GNAT file name rules should be considered as a tree file for the argument source file, gnatstub will refuse to create the tree file needed to create a body sampler, unless `-t` option is set

`-v` Verbose mode: gnatstub generates version information.

18 Minimizing Executables for Ada Programs Using gnatelim

18.1 About gnatelim

When you are working with a program which shares some set of Ada packages with other programs, it may happen, that this program uses only a part of subprogram defined in these packages, whereas the code created for unused subprograms is used for creating the executable and increases its size.

gnatelim is an utility tracking unused subprograms in an Ada program. Its output consists of a list of Eliminate pragmas marking all the subprograms that are declared, but never called in a given program. Eliminate is a GNAT-specific pragma, it is described in the next section. By placing the list of Eliminate pragmas in the GNAT configuration file 'gnat.adc' and recompiling your program, you may decrease the size of its executable, because the compiler will not create the code for unused subprograms.

The current version of gnatelim uses optimistic approach for some non-trivial cases arising during unused subprogram tracking, and as a result, for some programs you may have to correct a list of Eliminate pragmas manually.

gnatelim is an ASIS-based tool, and it needs as its input data a set of tree files representing all the components of a program to process. Itt also needs a bind file for a main subprogram. (See Section 18.3 [Preparing Tree and Bind Files for gnatelim], page 104 for full details)

18.2 Eliminate pragma

The syntax of Eliminate pragma is

```
pragma Eliminate (Library_Unit_Name, Subprogram_Name);
```

Library_Unit_Name
full expanded Ada name of a library unit

Subprogram_Name
a simple or expanded name of a subprogram declared within this compilation unit

The effect of an Eliminate pragma placed in the GNAT configuration file 'gnat.adc' is:

- If the subprogram denoted by Subprogram_Name is declared within the library unit having Library_Unit_Name as its defining program unit name, then the compiler will not create the code for this subprogram when compiling this unit, this applies to all overloaded subprograms denoted by Subprogram_Name.
- If a subprogram mentioned in some Eliminate pragma as unused is actually used (called) in a program, then the compiler will produce a diagnosis in place where it is called.

18.3 Preparing Tree and Bind Files for gnatelim

gnatelim can process only full Ada programs (partitions) and it needs a set of tree files representing the whole program (partition) to be presented in the current directory. It also needs a bind file for the main subprogram of the program (partition) to be presented in the current directory.

Let Main_Prog be the name of a main subprogram, and suppose this subprogram is in a file named 'main_prog.ads' or 'main_prog.adb'.

To create a minimal set of tree files covering the whole program, call gnatmake for this program as follows:

```
$ gnatmake -c [-f] -gnatc -gnatt Main_Prog
```

-c gnatmake option turns off the bind and link phases, which are impossible anyway, because sources are compiled with -gnatc option, which turns off generating object files.

If you have already done some compilations in your current directory, you should use -f gnatmake option, which forces recompilation of all the needed sources, even if they are up-to-date with the existing ALI files.

To create a bind file for gnatelim, run gnatbind for the main subprogram. gnatelim can work with either Ada or C bind file, if both are present, it works with the Ada bind file.

If you run gnatbind just after calling gnatmake for creating the tree files, or if there is some inconsistency between the existing object and source files, gnatbind will produce an error message, and you will have to reapply gnatmake, but without -gnatc -gnatt compiler options.

To avoid problems with creating a consistent data for gnatelim, you may use the following procedure. It creates all the data needed by gnatelim from scratch and therefore guarantees their consistency:

1. creating a bind file:

   ```
   $ gnatmake -c Main_Prog
   $ gnatbind main_prog.ali
   ```

2. creating a set of tree files:

   ```
   $ gnatmake -f -c -gnatc -gnatt Main_Prog
   ```

Note, that gnatelim needs neither object nor ALI files, so you may delete them at this stage.

18.4 Running gnatelim

gnatelim has the following command-line interface:

```
$ gnatelim [options] name
```

name should be a full expanded Ada name of a main subprogram of a program (partition).

gnatelim options:

-v Verbose mode: gnatelim version information is printed (in the form of Ada comments) to the standard output file. Various debugging information and information reflecting some details of the analysis doing by gnatelim are output to the standard error file.

gnatelim writes its output to the standard output file, so to write Eliminate pragmas produced by gnatelim into the GNAT configuration file 'gnat.adc', use the output redirection:

```
$ gnatelim Main_Prog > gnat.adc
```

or

```
$ gnatelim Main_Prog >> gnat.adc
```

If you would like to keep the existing contents of 'gnat.adc' and to append the gnatelim output to it.

18.5 Manual Correction of the List of Eliminate Pragmas

Created by gnatelim gnatelim

Sometimes gnatelim may try to eliminate subprograms which cannot really be eliminated because they are actually called in the program. (It is a very rare case for the current gnatelim version, but it still happens). In this case, the compiler will generate an error message of the form:

```
file.adb:106:07: cannot call eliminated subprogram "My_Prog"
```

You have to correct the 'gnat.adc' file manually by suppressing the faulty Eliminate pragmas. It is advised to recompile your program from scratch after that, because you need a consistent 'gnat.adc' file during the complete compilation in order to get an meaningful result.

18.6 Minimizing Your Executables

To get a smaller executable for your program, you have to recompile the program completely, having the 'gnat.adc' file with a set of Eliminate pragmas created by gnatelim in your current directory:

```
$ gnatmake -f Main_Prog
```

(you will need -f option for gnatmake to recompile everything with the set of pragmas Eliminate you have got from gnatelim).

Be aware, that a set of Eliminate pragmas is individual for every program. Therefore, you should never merge sets of Eliminate pragmas created for different programs in one 'gnat.adc' file.

18.7 Summary of the gnatelim Usage Cycle

Here is a summary of the steps you have to do to reduce the size of your executables with gnatelim. Note, that this is only an example, which creates all the data for gnatelim from scratch. You may try to reuse the tree file and the bind file when running gnatelim several times for the same program. You may also use other GNAT options (you may to control the optimization level, produce the debugging information, set search path etc).

1. producing a set of tree files and a bind file:

```
$ gnatmake -c Main_Prog
$ gnatbind main_prog.ali
$ gnatmake -f -c -gnatc -gnatt Main_Prog
```

2. generating a list of `Eliminate` pragmas

   ```
   $ gnatelim Main_Prog >[>] gnat.adc
   ```

3. recompiling everything with '`gnat.adc`' file to create a smaller executable (manual correction of '`gnat.adc`' may be needed):

   ```
   $ gnatmake -f Main_Prog
   ```

19 Other Utility Programs

This chapter discusses some other utility programs available in the Ada environment.

19.1 Using Other Utility Programs With GNAT

The object files generated by GNAT are in standard system format and in particular the debugging information uses this format. This means programs generated by GNAT can be used with existing utilities that depend on these formats.

In general, any utility program that works with C will also often work with Ada programs generated by GNAT. This includes software utilities such as gprof (a profiling program), gdb (the FSF debugger), and utilities such as Purify.

19.2 The `gnatpsys` Utility Program

Many of the definitions in package System are implementation-dependent. Furthermore, although the source of the package System is available for inspection, it uses special attributes for parametrizing many of the critical values, so the source is not informative for the casual user.

The `gnatpsys` utility is designed to deal with this situation. It is an Ada program that dynamically determines the values of all the relevant parameters in System, and prints them out in the form of an Ada source listing for System, displaying all the values of interest. This output is generated to '`stdout`'.

To determine the value of any parameter in package System, simply run `gnatpsys` with no qualifiers or arguments, and examine the output. This is preferable to consulting documentation, because you know that the values you are getting are the actual ones provided by the executing system.

19.3 The `gnatpsta` Utility Program

Many of the definitions in package Standard are implementation-dependent. However, the source of this package does not exist as an Ada source file, so these values cannot be determined by inspecting the source. They can be determined by examinining in detail the coding of '`cstand.adb`' which creates the image of Standard in the compiler, but this is awkward and requires a great deal of internal knowledge about the system.

The `gnatpsta` utility is designed to deal with this situation. It is an Ada program that dynamically determines the values of all the relevant parameters in Standard, and prints them out in the form of an Ada source listing for Standard, displaying all the values of interest. This output is generated to '`stdout`'.

To determine the value of any parameter in package Standard, simply run `gnatpsta` with no qualifiers or arguments, and examine the output. This is preferable to consulting documentation, because you know that the values you are getting are the actual ones provided by the executing system.

19.4 The External Symbol Naming Scheme of GNAT

In order to interpret the output from GNAT, when using tools that are originally intended for use with other languages, it is useful to understand the conventions used to generate link names from the Ada entity names.

All link names are in all lowercase letters. With the exception of library procedure names, the mechanism used is simply to use the full expanded Ada name with dots replaced by double underscores. For example, suppose we have the following package spec:

```
package QRS is MN : Integer; end QRS;
```

The variable `MN` has a full expanded Ada name of `QRS.MN`, so the corresponding link name is `qrs__mn`. Of course if a `pragma Export` is used this may be overridden:

```
package Exports is
   Var1 : Integer;
   pragma Export (Var1, C, External_Name => "var1_name");
   Var2 : Integer;
   pragma Export (Var2, C, Link_Name => "var2_link_name");
end Exports;
```

In this case, the link name for *Var1* is *var1_name*, and the link name for *Var2* is *var2_link_name*.

One exception occurs for library level procedures. A potential ambiguity arises between the required name `_main` for the C main program, and the name we would otherwise assign to an Ada library level procedure called `Main` (which might well not be the main program).

To avoid this ambiguity, we attach the prefix `_ada_` to such names. So if we have a library level procedure such as

```
procedure Hello (S : String);
```

the external name of this procedure will be *_ada_hello*.

19.5 Ada Mode for emacs

The Emacs mode for programming in Ada (both, Ada83 and Ada95) helps the user in understanding existing code and facilitates writing new code. It furthermore provides some utility functions for easier integration of standard Emacs features when programming in Ada.

19.6 General features:

- full Integrated Development Environment :
  - support of 'project files' for the configuration (directories, compilation options,...)
  - compiling and stepping through error messages.
  - running and debugging your applications within Emacs.
- easy to use for beginners by pull-down menus,
- user configurable by many user-option variables.

19.7 Ada mode features that help understanding code:

- functions for easy and quick stepping through Ada code,
- getting cross reference information for identifiers (e.g. find the defining place by a keystroke),
- displaying an index menu of types and subprograms and move point to the chosen one,
- automatic color highlighting of the various entities in Ada code.

19.8 Emacs support for writing Ada code:

- switching between spec and body files with eventually autogeneration of bodyfiles,
- automatic formating of subprograms parameter lists.
- automatic smart indentation according to Ada syntax,
- automatic completion of identifiers,
- automatic casing of identifiers, keywords, and attributes,
- insertion of statement templates,
- filling comment paragraphs like filling normal text,

Please see See Section 19.5 [Ada Mode for emacs], page 108 for more information.

19.9 Converting Ada files to html using ada2html

This `Perl` script allows Ada source files to be browsed using standard Web browsers. See the section See Section 19.10 [Installing ada2html], page 110 for installation procedure.

Ada reserved keywords are highlighted in a bold font and Ada comments in a blue font. If your program was compiled using the `Gnat` -gnatx2 switch to generate cross-referencing information, user defined variables and types will appear in a different color. You will be able to click on any identifier and go to its declaration, as long as you used the -gnatx2 switch.

The command line is as follow:

```
$ ada2html [switches] ada-files
```

You can pass it as many Ada files as you want. `ada2html` will generate an html file for every ada file, and a global file called '`index.htm`'. This file is an index of every identifier defined in the files.

The available switches are the following ones :

`-83` Only the subset on the Ada 83 keywords will be highlighted, not the full Ada 95 keywords set.

`-d` If the ada files depend on some other files (using for instance the `with` command, the latter will also be converted to html.

`-l`*number* If this switch is provided and *number* is not 0, then `ada2html` will number the html files every *number* line.

`-p`*file* If you are using Emacs and the most recent Emacs Ada mode, which provides a full Integrated Development Environment for compiling, checking, running and debugging applications, you may be using '`.prj`' files to give the directories where Emacs can find sources and object files.

Using this switch, you can tell ada2html to use these files. This allows you to get an html version of your application, even if it is spread over multiple directories.

19.10 Installing ada2html

`Perl` needs to be installed on your machine to run this script. `Perl` is freely available for almost every architecture and Operating System via the internet.

On Unix systems, you may have to modify the first line of the script `ada2html`, to explicitly tell the Operating system where Perl is. The syntax of this line is :

```
#!full_path_name_to_perl
```

Alternatively, you may run the script using the following command line:

```
perl ada2html [switches] files
```

20 Running and Debugging Ada Programs

This chapter discusses how to debug Ada programs. An incorrect Ada program may be handled in three ways by the GNAT compiler:

1. The illegality may be a violation of the static semantics of Ada. In that case GNAT diagnoses the constructs in the program that are illegal. It is then a straighforward matter for the user to modify those parts of the program.
2. The illegality may be a violation of the dynamic semantics of Ada. In that case the program compiles and executes, but may generate incorrect results, or may terminate abnormally with some exception.
3. When presented with a program that contains convoluted errors, GNAT itself may terminate abnormally without providing full diagnostics on the incorrect user program.

20.1 The GNAT Debugger GDB

GDB is a general purpose, platform-independent debugger that can be used to debug mixed-language programs compiled with GCC, and in particular is capable of debugging Ada programs compiled with GNAT. The latest versions of GDB are Ada-aware and can handle complex Ada data structures. The manual *Debugging with GDB* contains full details on the usage of GDB, including a section on its usage on programs. This manual should be consulted for full details. The section that follows is a brief introduction to the philosophy and use of GDB.

When GNAT programs are compiled, the compiler optionally writes debugging information into the generated object file, including information on line numbers, and on declared types and variables. This information is separate from the generated code. It makes the object files considerably larger, but it does not add to the size of the actual executable that will be loaded into memory, and has no impact on run-time performance. The generation of debug information is triggered by the use of the -g switch in the gcc or gnatmake command used to carry out the compilations. It is important to emphasize that the use of these options does not change the generated code.

The debugging information is written in standard system formats that are used by many tools, including debuggers and profilers. The format of the information is typically designed to describe C types and semantics, but GNAT implements a translation scheme which allows full details about Ada types and variables to be encoded into these standard C formats. Details of this encoding scheme may be found in the file exp_dbug.ads in the GNAT source distribution. However, the details of this encoding are, in general, of no interest to a user, since GDB automatically performs the necessary decoding.

When a program is bound and linked, the debugging information is collected from the object files, and stored in the executable image of the program. Again, this process significantly increases the size of the generated executable file, but it does not increase the size of the executable program itself. Furthermore, if this program is run in the normal manner, it runs exactly as if the debug information were not present, and takes no more actual memory.

However, if the program is run under control of GDB, the debugger is activated. The image of the program is loaded, at which point it is ready to run. If a run command is given, then the program will run exactly as it would have if GDB were not present. This is a crucial part of the GDB design philsophy. GDB is entirely non-intrusive until a breakpoint is encountered. If no breakpoint is ever hit, the program will run exactly as it would if no debugger were present. When a breakpoint is hit, GDB accesses the debugging information and can respond to user commands to inspect variables, and more generally to report on the state of execution.

20.2 Running GDB

The command to run GDB is

```
gdb program
```

where `program` is the name of the executable file. This activates the debugger and results in a prompt for debugger commands. The simplest command is simply `run`, which causes the program to run exactly as if the debugger were not present. The following section describes some of the additional commands that can be given to GDB.

20.3 Introduction to GDB Commands

GDB contains a large repertoire of commands. The manual *Debugging with GDB* includes extensive documentation on the use of these commands, together with examples of their use. Furthermore, the command *help* invoked from within GDB activates a simple help facility which summarizes the available commands and their options. In this section we summarize a few of the most commonly used commands to give an idea of what GDB is about. You should create a simple program with debugging information and experiment with the use of these GDB commands on the program as you read through the following section.

`set args` *arguments*

The *arguments* list above is a list of arguments to be passed to the program on a subsequent run command, just as though the arguments had been entered on a normal invocation of the program. The `set args` command is not needed if the program does not require arguments.

`run`

The `run` command causes execution of the program to start from the beginning. If the program is already running, that is to say if you are currently positioned at a breakpoint, then a prompt will ask for confirmation that you want to abandon the current execution and restart.

`breakpoint` *location*

The breakpoint command sets a breakpoint, that is to say a point at which execution will halt and GDB will await further commands. *location* is either a line number within a file, given in the format `file:linenumber`, or it is the name of a subprogram. If you request that a breakpoint be set on a subprogram that is overloaded, a prompt will ask you to specify on which of those subprograms you want to breakpoint. You can also specify that all of them should be breakpointed. If the program is run and execution encounters the breakpoint, then the program stops and GDB signals that the breakpoint was encountered by printing the line of code before which the program is halted.

`breakpoint exception` *name*
: A special form of the breakpoint command which breakpoints whenever exception *name* is raised. If *name* is omitted, then a breakpoint will occur when any exception is raised.

`print` *expression*
: This will print the value of the given expression. Most simple Ada expression formats are properly handled by GDB, so the expression can contain function calls, variables, operators, and attribute references.

`continue`
: Continues execution following a breakpoint, until the next breakpoint or the termination of the program.

`step`
: Executes a single line after a breakpoint. If the next statement is a subprogram call, execution continues into (the first statement of) the called subprogram.

`next`
: Executes a single line. If this line is a subprogram call, executes and returns from the call.

`list`
: Lists a few lines around the current source location. In practice, it is usually more convenient to have a separate edit window open with the relevant source file displayed. Successive applications of this command print subsequent lines. The command can be given an argument which is a line number, in which case it displays a few lines around the specified one.

`backtrace`
: Displays a backtrace of the call chain. This command is typically used after a breakpoint has occured, to examine the sequence of calls that leads to the current breakpoint. The display includes one line for each activation record (frame) corresponding to an active subprogram.

`up`
: At a breakpoint, GDB can display the values of variables local to the current frame. The command `up` can be used to examine the contents of other active frames, by moving the focus up the stack, that is to say from callee to caller, one frame at a time.

`down`
: Moves the focus of GDB down from the frame currently being examined to the frame of its callee (the reverse of the previous command),

`frame` *n*
: Inspect the frame with the given number. The value 0 denotes the frame of the current breakpoint, that is to say the top of the call stack.

The above list is a very short introduction to the commands that GDB provides. Important additional capabilities, including conditional breakpoints, the ability to execute command sequences on a breakpoint, the ability to debug at the machine instruction level and many other features are described in detail in *Debugging with GDB*. Note that most commands can be abbreviated (for example, c for continue, bt for backtrace).

20.4 Using Ada Expressions

GDB supports a fairly large subset of Ada expression syntax, with some extensions. The philosophy behind the design of this subset is

- That GDB should provide basic literals and access to operations for arithmetic, dereferencing, field selection, indexing, and subprogram calls, leaving more sophisticated computations to subprograms written into the program (which therefore may be called from GDB).
- That type safety and strict adherence to Ada language restrictions are not particularly important to the GDB user.
- That brevity is important to the GDB user.

Thus, for brevity, the debugger acts as if there were implicit `with` and `use` clauses in effect for all user-written packages, thus making it unnecessary to fully qualify most names with their packages, regardless of context. Where this causes ambiguity, GDB asks the user's intent.

For details on the supported Ada syntax see Section 20.1 [The GNAT Debugger GDB], page 111.

20.5 Calling User-Defined Subprograms

An important capability of GDB is the ability to call user-defined subprograms while debugging. This is achieved simply by entering a subprogram call statement in the form:

```
call subprogram-name (parameters)
```

The keyword `call` can be omitted in the normal case where the `subprogram-name` does not coincide with any of the predefined GDB commands.

The effect is to invoke the given subprogram, passing it the list of parameters that is supplied. The parameters can be expressions and can include variables from the program being debugged. The subprogram must be defined at the library level within your program, and GDB will call the subprogram within the environment of your program execution (which means that the subprogram is free to access or even modify variables within your program).

The most important use of this facility is in allowing the inclusion of debugging routines that are tailored to particular data structures in your program. Such debugging routines can be written to provide a suitably high-level description of an abstract type, rather than a low-level dump of its physical layout. After all, the standard GDB `print` command only knows the physical layout of your types, not their abstract meaning. Debugging routines can provide information at the desired semantic level and are thus enormously useful.

For example, when debugging GNAT itself, it is crucial to have access to the contents of the tree nodes used to represent the program internally. But tree nodes are represented simply by an integer value (which in turn is an index into a table of nodes). Using the `print` command on a tree node would simply print this integer value, which is not very useful. But the PN routine (defined in file treepr.adb in the GNAT sources) takes a tree node as input, and displays a useful high level representation of the tree node, which includes the syntactic category of the node, its position in the source, the integers that denote descendant nodes and parent node, as well as varied semantic information. To study this example in more detail, you might want to look at the body of the PN procedure in the stated file.

20.6 Breaking on Ada Exceptions

You can set breakpoints that trip when your program raises selected exceptions.

`break exception`

Set a breakpoint that trips whenever (any task in the) program raises any exception.

`break exception` *name*

Set a breakpoint that trips whenever (any task in the) program raises the exception *name*.

`break exception unhandled`

Set a breakpoint that trips whenever (any task in the) program raises an exception for which there is no handler.

`info exceptions`
`info exceptions` *regexp*

The `info exceptions` command permits the user to examine all defined exceptions within Ada programs. With a regular expression, *regexp*, as argument, prints out only those exceptions whose name matches *regexp*.

20.7 Support for Ada Tasks

GDB allows the following task-related commands:

`info tasks`

This command shows a list of current Ada tasks, as in the following example:

```
(gdb) info tasks
  ID       TID P-ID   Thread Pri State                 Name
   1   8088000   0   807e000  15 Child Activation Wait main_task
   2   80a4000   1   80ae000  15 Accept/Select Wait    b
   3   809a800   1   80a4800  15 Child Activation Wait a
*  4   80ae800   3   80b8000  15 Running               c
```

In this listing, the asterisk before the first task indicates it to be the currently running task. The first column lists the task ID that is used to refer to tasks in the following commands.

`break` *linespec* `task` *taskid*
`break` *linespec* `task` *taskid* `if` ...

These commands are like the `break` ... `thread` .... *linespec* specifies source lines.

Use the qualifier '`task` *taskid*' with a breakpoint command to specify that you only want GDB to stop the program when a particular Ada task reaches this breakpoint. *taskid* is one of the numeric task identifiers assigned by GDB, shown in the first column of the '`info tasks`' display.

If you do not specify '`task` *taskid*' when you set a breakpoint, the breakpoint applies to *all* tasks of your program.

You can use the `task` qualifier on conditional breakpoints as well; in this case, place '`task` *taskid*' before the breakpoint condition (before the `if`).

For more detailed information on the tasking support see Section 20.1 [The GNAT Debugger GDB], page 111.

20.8 Debugging Generic Units

GNAT always uses code expansion for generic instantiation. This means that each time an instantiation occurs, a complete copy of the original code is made, with appropriate substitutions of formals by actuals.

It is not possible to refer to the original generic entities in GDB, but it is always possible to debug a particular instance of a generic, by using the appropriate expanded names. For example, if we have

```
procedure g is

   generic package k is
      procedure kp (v1 : in out integer);
   end k;

   package body k is
      procedure kp (v1 : in out integer) is
      begin
         v1 := v1 + 1;
      end kp;
   end k;

   package k1 is new k;
   package k2 is new k;

   var : integer := 1;

begin
   k1.kp (var);
   k2.kp (var);
   k1.kp (var);
   k2.kp (var);
end;
```

Then to break on a call to procedure kp in the k2 instance, simply use the command:

```
break g.k2.kp
```

When the breakpoint occurs, you can step through the code of the instance in the normal manner and examine the values of local variables, as for other units.

20.9 GNAT Abnormal Termination

When presented with programs that contain serious errors in syntax or semantics, GNAT may on rare occasions experience problems in operation, such as aborting with a segmentation fault or illegal memory access, raising an internal exception, or terminating abnormally. In such cases, you can activate various features of GNAT that can help you pinpoint the construct in your program that is the likely source of the problem.

The following strategies are presented in increasing order of difficulty, corresponding to your programming skills and your familiarity with compiler internals.

1. Run `gcc` with the `-gnatf` and `-gnate` switches. The first switch causes all errors on a given line to be reported. In its absence, only the first error on a line is displayed.

   The `-gnate` switch causes errors to be displayed as soon as they are encountered, rather than after compilation is terminated. If GNAT terminates prematurely, the last error message displayed is likely to pinpoint the culprit.
2. Run `gcc` with the `-v (verbose)` switch. In this mode, `gcc` produces ongoing information about the progress of the compilation and provides the name of each procedure as code is generated. This switch allows you to find which Ada procedure was being compiled when it encountered a code generation problem.
3. Run `gcc` with the `-gnatdc` switch. This is a GNAT specific switch that does for the front-end what `-v` does for the backend. The system prints the name of each unit, either a compilation unit or nested unit, as it is being analyzed.
4. Finally, you can start `gdb` directly on the `gnat1` executable. `gnat1` is the front-end of GNAT, and can be run independently (normally it is just called from `gcc`). You can use `gdb` on `gnat1` as you would on a C program (but see Section 20.1 [The GNAT Debugger GDB], page 111 for caveats). The `where` command is the first line of attack; the variable `lineno` (seen by `print lineno`), used by the second phase of `gnat1` and by the `gcc` backend, indicates the source line at which the execution stopped, and `input_file name` indicates the name of the source file.

20.10 Naming Conventions for GNAT Source Files

In order to examine the workings of the GNAT system, the following brief description of its organization may be helpful:

- Files with prefix '`sc`' contain the lexical scanner.
- All files prefixed with '`par`' are components of the parser. The numbers correspond to chapters of the Ada 95 Reference Manual. For example, parsing of select statements can be found in '`par-ch9.adb`'.
- All files prefixed with '`sem`' perform semantic analysis. The numbers correspond to chapters of the Ada standard. For example, all issues involving context clauses can be found in '`sem_ch10.adb`'. In addition, some features of the language require sufficient special processing to justify their own semantic files: sem_aggr for aggregates, sem_disp for dynamic dispatching, etc.
- All files prefixed with '`exp`' perform normalization and expansion of the intermediate representation (abstract syntax tree, or AST). these files use the same numbering scheme as the parser and semantics files. For example, the construction of record initialization procedures is done in '`exp_ch3.adb`'.
- The files prefixed with '`bind`' implement the binder, which verifies the consistency of the compilation, determines an order of elaboration, and generates the bind file.
- The files '`atree.ads`' and '`atree.adb`' detail the low-level data structures used by the front-end.

- The files 'sinfo.ads' and 'sinfo.adb' detail the structure of the abstract syntax tree as produced by the parser.
- The files 'einfo.ads' and 'einfo.adb' detail the attributes of all entities, computed during semantic analysis.
- Library management issues are dealt with in files with prefix 'lib'.
- Ada files with the prefix 'a-' are children of Ada, as defined in Annex A.
- Files with prefix 'i-' are children of Interfaces, as defined in Annex B.
- Files with prefix 's-' are children of System. This includes both language-defined children and GNAT run-time routines.
- Files with prefix 'g-' are children of GNAT. These are useful general-purpose packages, fully documented in their specifications. All the other '.c' files are modifications of common gcc files.

20.11 Getting Internal Debugging Information

Most compilers have internal debugging switches and modes. GNAT does also, except GNAT internal debugging switches and modes are not secret. A summary and full description of all the compiler and binder debug flags are in the file 'debug.adb'. You must obtain the sources of the compiler to see the full detailed effects of these flags.

The switches that print the source of the program (reconstructed from the internal tree) are of general interest for user programs, as are the options to print the full internal tree, and the entity table (the symbol table information). The reconstructed source provides a readable version of the program after the front-end has completed analysis and expansion, and is useful when studying the performance of specific constructs. For example, constraint checks are indicated, complex aggregates are replaced with loops and assignments, and tasking primitives are replaced with run-time calls.

21 Performance Considerations

The GNAT system provides a number of options that allow a trade-off between

- performance of the generated code
- speed of compilation
- minimization of dependences and recompilation
- the degree of run-time checking.

The defaults (if no options are selected) aim at improving the speed of compilation and minimizing dependences, at the expense of performance of the generated code:

- no optimization
- no inlining of subprogram calls
- all run-time checks enabled except overflow and elaboration checks

These options are suitable for most program development purposes. This chapter describes how you can modify these choices.

21.1 Controlling Run-time Checks

By default, GNAT produces all run-time checks, except arithmetic overflow checking for integer operations (that includes division by zero) and checks for access before elaboration on subprogram calls. Two gnat switches, `-gnatp` and `-gnato` allow this default to be modified. See Section 3.2.3 [Run-time Checks], page 27.

Our experience is that the default is suitable for most development purposes.

We treat integer overflow and elaboration checks specially because these are quite expensive and in our experience are not as important as other run-time checks in the development process.

Note that the setting of the switches controls the default setting of the checks. They may be modified using either `pragma Suppress` (to remove checks) or `pragma Unsuppress` (to add back suppressed checks) in the program source.

21.2 Optimization Levels

The default is optimization off. This results in the fastest compile times, but GNAT makes absolutely no attempt to optimize, and the generated programs are considerably larger and slower than when optimization is enabled. You can use the `-O`*n* switch, where *n* is an integer from 0 to 3, on the `gcc` command line to control the optimization level:

`-O0` no optimization (the default)

`-O1` medium level optimization

`-O2` full optimization

`-O3` full optimization, and also attempt automatic inlining of small subprograms within a unit (see Section 21.3 [Inlining of Subprograms], page 120).

Higher optimization levels perform more global transformations on the program and apply more expensive analysis algorithms in order to generate faster and more compact code. The price in compilation time, and the resulting improvement in execution time, both depend on the particular application and the hardware environment. You should experiment to find the best level for your application.

Note: Unlike some other compilation systems, `gcc` has been tested extensively at all optimization levels. There are some bugs which appear only with optimization turned on, but there have also been bugs which show up only in *unoptimized* code. Selecting a lower level of optimization does not improve the reliability of the code generator, which in practice is highly reliable at all optimization levels.

21.3 Inlining of Subprograms

A call to a subprogram in the current unit is inlined if all the following conditions are met:

- The optimization level is at least `-O1`.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else that `gcc` cannot support in inlined subprograms.
- The call occurs after the definition of the body of the subprogram.
- Either `pragma Inline` applies to the subprogram or it is small and automatic inlining (optimization level `-O3`) is specified.

Calls to subprograms in `with`'ed units are normally not inlined. To achieve this level of inlining, the following conditions must all be true:

- The optimization level is at least `-O1`.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else `gcc` cannot support in inlined subprograms.
- The call appears in a body (not in a package spec).
- There is a `pragma Inline` for the subprogram.
- The `-gnatn` switch is used in the `gcc` command line

Note that specifying the `-gnatn` switch causes additional compilation dependencies. Consider the following:

```
package R is
   procedure Q;
   pragma Inline Q;
end R;
package body R is
   ...
end R;
with R;
procedure Main is
begin
   ...
   R.Q;
end Main;
```

With the default behavior (no -gnatn switch specified), the compilation of the Main procedure depends only on its own source, 'main.adb', and the spec of the package in file 'r.ads'. This means that editing the body of R does not require recompiling Main.

On the other hand, the call R.Q is not inlined under these circumstances. If the -gnatn switch is present when Main is compiled, the call will be inlined if the body of Q is small enough, but now Main depends on the body of R in 'r.adb' as well as on the spec. This means that if this body is edited, the main program must be recompiled. Note that this extra dependency occurs whether or not the call is in fact inlined by gcc.

Note: The -fno-inline switch can be used to prevent all inlining. This switch overrides all other conditions and ensures that no inlining occurs. The extra dependences resulting from -gnatn will still be active, even if this switch is used to suppress the resulting inlining actions.

22 Windows NT and Windows95 Topics

This chapter describes topics that are specific to the Windows platforms.

22.1 Introduction to Dynamic Link Libraries (DLLs)

This section gives a minimal introduction on using dynamic link libraries. A DLL is a library that can be shared by many programs, it contains a set of Ada compilation units (e.g. packages or top-level subprograms) or C functions.

The Windows program loader is in charge of actually linking the program to the DLLs it uses. In a nutshell, the loader first maps the program into memory, and then loads all DLLs that the program uses and connects them to the program. At the time the program is connected to the DLL, the loader maps the data into the program's address space (so the code is shared but the data is not).

A program using DLLs has to contain extra information for all dynamic links that must be resolved by the loader. This information is usually contained in an import library. Due to limitations in the loader it is only possible to link with functions in a DLL. If you need to access a variable in a DLL, the best solution is to write a wrapper function.

22.2 Using DLLs With GNAT

This section describes how to link with a dynamic link library containing C or Ada code using the GNAT compiler on the Windows NT or Windows 95 platform. The example presented here assumes the DLL is named 'API.DLL' and comes with an import library ('.lib') built with MicroSoft tools.

The MicroSoft libraries are not completely compatible with the GNAT linker, so it is necessary to build an import library for use with GNAT. The GNAT linker uses the Unix naming scheme for linking with libraries: the import library for 'API.DLL' is called 'libAPI.a' or 'libapi.a'. (Names are not case sensitive.)

Here are the steps to create a GNAT import library named 'libapi.a' for the 'API.DLL' example and to link a program with the DLL using this library.

1. Create the definition file from the DLL

   ```
   $ dll2def api.dll > api.def
   ```

2. Build the import library itself

   ```
   $ dlltool --dllname api.dll --def api.def --output-lib libapi.a
   ```

3. Writing the Import clauses

   When using C there are usually header files that contain definitions of types, function profiles and constants exported by the DLL. The Ada equivalent for these declarations are typically defined in a separate package specification that contains only named numbers, type definitions and imported subprograms. This is often called a thin binding.

   Assume that for the 'API.DLL' there is a file 'api.h' containing just the following function prototype:

```
int get (char *);
```

In this case the stub Ada package will be:

```
with Interfaces.C, Interfaces.C.Strings;
package API is

   function Get (C : in Interfaces.C.Strings.Chars_Ptr)
      return Interfaces.C.Int;

private

   pragma Import (C, Get);

end API;
```

To link against this DLL add the arguments '-largs -lapi' to the gnatmake command line or, alternatively, add 'pragma Linker_Options ("-lapi")' to the API package.

```
$ gnatmake my_main -largs -lapi
```

The GNU linker option '-l<name>' causes the import library 'lib<name>.a' to be included in the link.

See the next section for a discussion about calling conventions.

22.3 Windows 95/NT calling conventions

Most DLLs use the C or Stdcall calling conventions, the latter is Windows specific. For other calling conventions see Section 2.10 [Mixed Language Programming], page 17.

- C

  This convention is used when interfacing to the C language.

  The function parammeters are pushed on the stack from right to left, and the caller is in charge of cleaning up the stack.

  In C, the name the symbol defined during the compilation is the name of the C function with an underscore at the start.

  The C function:

  ```
  int get_val (long);
  ```

  Should be imported from Ada with:

  ```
  function Get_Val (V : in Interfaces.C.Long)
     return Interfaces.C.Int;
  pragma Import (C, Get_Val);
  ```

  The symbol defined here is '_get_val'.

- Stdcall

  This convention is used to interface to the Win32 API.

  The function parammeters are pushed on the stack from right to left, and the called function cleans up the stack on function exit.

The defined symbol is obtained from the function name by prefixing an underscore, and appending the characters `@` followed by the number of bytes occupied by the parameters.

The Win32 function:

```
 APIENTRY int get_val (long);
```

Should be called from Ada with:

```
function Get_Val (V : in Interfaces.C.Long)
  return Interfaces.C.Int;

pragma Import (Stdcall, Get_Val);
```

The symbol defined here is '`_get_val@4`', given that a long is defined with four bytes under Windows 95 and Windows NT 4.0.

The name of the function can be specified by means of the third parameter (External_Name) of the pragma Import.

```
function Get_Val (V : in Interfaces.C.Long)
return Interfaces.C.Int;

pragma Import (Stdcall, Get_Val, "retrieve_val");
```

Here the defined symbol is `_retrieve_val@4`.

The name of the symbol can be specified with the fourth parameter (Link_Name) of the pragma Import.

```
function Get_Val (V : in Interfaces.C.Long)
return Interfaces.C.Int;

pragma Import (Stdcall, Get_Val, Link_Name => "retrieve_val");
```

Here the defined symbol is `retrieve_val@4` (there is no underscore at the start).

Index

(Index is nonexistent)

Table of Contents

About This Guide 1

What This Guide Contains 1
What You Should Know Before Reading This Guide 2
Related Information 2
Conventions 2

1 Getting Started With GNAT 3

1.1 Running GNAT 3
1.2 Running a Simple Ada Program 3
1.3 Running a Program With Multiple Units 4
1.4 Using the `gnatmake` Utility 5

2 The GNAT Compilation Model 7

2.1 Source Representation 7
2.2 Foreign Language Representation 7
2.2.1 Latin-1 8
2.2.2 Other 8-Bit Codes 8
2.2.3 Wide Character Encodings 9
2.3 File Naming Rules 10
2.4 Using Other File Names 11
2.5 Generating Object Files 12
2.6 Source Dependencies 12
2.7 The Ada Library Information Files 13
2.8 Representation of Time Stamps 16
2.9 Binding an Ada Program 16
2.10 Mixed Language Programming 17
2.11 Comparison between GNAT and C/C++ Compilation Models 18
2.12 Comparison between GNAT and Conventional Ada Library Models 19

3 Compiling Using `gcc` 21

3.1 Compiling Programs 21
3.2 Switches for `gcc` 22
3.2.1 Error Message Control 24
3.2.2 Debugging and Assertion Control 27
3.2.3 Run-time Checks 27
3.2.4 Using `gcc` for Syntax Checking 28
3.2.5 Using `gcc` for Semantic Checking 28
3.2.6 Compiling Ada 83 Programs 29
3.2.7 Style Checking 29

3.2.8 Character Set Control 30
3.2.9 File Naming Control 31
3.2.10 Subprogram Inlining Control 31
3.2.11 Auxiliary Output Control 31
3.2.12 Debugging Control 32
3.3 Search Paths and the Run-Time Library (RTL) 33
3.4 Order of Compilation Issues 34
3.5 Examples 34

4 Binding Using gnatbind 35
4.1 Running gnatbind 35
4.2 Consistency-Checking Modes 38
4.3 Binder Error Message Control 38
4.4 Elaboration Control 39
4.5 Output Control 39
4.6 Binding for Non-Ada Main Programs 39
4.7 Summary of Binder Switches 40
4.8 Command-Line Access 41
4.9 Search Paths for gnatbind 41
4.10 Examples of gnatbind Usage 42

5 Linking Using gnatlink 43
5.1 Running gnatlink 43
5.2 Switches for gnatlink 43

6 The GNAT Make Program gnatmake 45
6.1 Running gnatmake 45
6.2 Switches for gnatmake 46
6.3 Mode switches for gnatmake 48
6.4 Notes on the Command Line 49
6.5 How gnatmake Works 50
6.6 Examples of gnatmake Usage 50

7 Renaming Files Using gnatchop 51
7.1 Handling Files with Multiple Units 51
7.2 Operation gnatchop in Compilation Mode 51
7.3 Command Line for gnatchop 52
7.4 Switches for gnatchop 53
7.5 Examples of gnatchop Usage 54

8 Handling of Configuration Pragmas 55
8.1 The gnat.adc file 55

9 Elaboration Order Handling in GNAT 57
- 9.1 Elaboration Code in Ada 95 57
- 9.2 Checking the Elaboration Order in Ada 95 58
- 9.3 Controlling the Elaboration Order in Ada 95 59
- 9.4 Controlling Elaboration in GNAT - Internal Calls 62
- 9.5 Controlling Elaboration in GNAT - External Calls 65
- 9.6 Default Behavior in GNAT - Ensuring Safety 66
- 9.7 What to do if the Default Elaboration Behavior Fails 67
- 9.8 Elaboration for Access-to-Subprogram Values 69
- 9.9 Summary of Procedures for Elaboration Control 69

10 The cross-referencing tools `gnatxref` and `gnatfind` 71
- 10.1 Common switches 71
- 10.2 Gnatxref switches 72
- 10.3 gnatfind switches 72
- 10.4 Project files 73
- 10.5 Regular expressions in gnatfind and gnatxref 74
- 10.6 Examples of `gnatxref` usage 75
  - 10.6.1 General usage 75
  - 10.6.2 Using gnatxref with vi 76
- 10.7 Examples of `gnatfind` usage 76

11 File Name Krunching Using `gnatkr` 79
- 11.1 About `gnatkr` 79
- 11.2 Using `gnatkr` 79
- 11.3 Krunching Method 80
- 11.4 Examples of `gnatkr` Usage 81

12 Preprocessing Using `gnatprep` 83
- 12.1 Using `gnatprep` 83
- 12.2 Switches for `gnatprep` 83
- 12.3 Form of definitions file 84
- 12.4 Form of input text for `gnatprep` 84

13 The GNAT run-time library builder `gnatlbr` 87
- 13.1 Running `gnatlbr` 87
- 13.2 Switches for `gnatlbr` 87
- 13.3 Example of `gnatlbr` Usage 87

14 The GNAT library browser `gnatls` 89
- 14.1 Running `gnatls` 89
- 14.2 Switches for `gnatls` 90
- 14.3 Example of `gnatls` Usage 91

15 Finding memory problems with gnatmem .. 93
15.1 Running gnatmem 93
15.2 Switches for gnatmem 94
15.3 Example of gnatmem Usage 94
15.4 Implementation note 97

16 ASIS-Based Tools 99
16.1 The ASIS Implementation for GNAT (ASIS-for-GNAT) .. 99
16.2 Tree Files 99

17 Creating Sample Bodies Using gnatstub .. 101
17.1 Running gnatstub 101
17.2 Switches for gnatstub 101

18 Minimizing Executables for Ada Programs Using gnatelim 103
18.1 About gnatelim 103
18.2 Eliminate pragma 103
18.3 Preparing Tree and Bind Files for gnatelim 104
18.4 Running gnatelim 104
18.5 Manual Correction of the List of Eliminate Pragmas .. 105
18.6 Minimizing Your Executables 105
18.7 Summary of the gnatelim Usage Cycle 105

19 Other Utility Programs 107
19.1 Using Other Utility Programs With GNAT 107
19.2 The gnatpsys Utility Program 107
19.3 The gnatpsta Utility Program 107
19.4 The External Symbol Naming Scheme of GNAT 108
19.5 Ada Mode for emacs 108
19.6 General features: 108
19.7 Ada mode features that help understanding code: 109
19.8 Emacs support for writing Ada code: 109
19.9 Converting Ada files to html using ada2html 109
19.10 Installing ada2html 110

20 Running and Debugging Ada Programs . . 111
20.1 The GNAT Debugger GDB 111
20.2 Running GDB 112
20.3 Introduction to GDB Commands 112
20.4 Using Ada Expressions 113
20.5 Calling User-Defined Subprograms 114
20.6 Breaking on Ada Exceptions 115
20.7 Support for Ada Tasks 115
20.8 Debugging Generic Units 116
20.9 GNAT Abnormal Termination 116
20.10 Naming Conventions for GNAT Source Files 117
20.11 Getting Internal Debugging Information 118

21 Performance Considerations 119
21.1 Controlling Run-time Checks 119
21.2 Optimization Levels 119
21.3 Inlining of Subprograms 120

22 Windows NT and Windows95 Topics 123
22.1 Introduction to Dynamic Link Libraries (DLLs) 123
22.2 Using DLLs With GNAT 123
22.3 Windows 95/NT calling conventions 124

Index 127