

# GNAT User's Guide for Native Platforms

---

GNAT User's Guide for Native Platforms , Dec 05, 2025

AdaCore

Copyright © 2008-2025, Free Software Foundation

---















‘GNAT, The GNU Ada Development Environment’

GCC version 16.0.0

AdaCore

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









```
end Greetings;

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

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

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

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

```
'greetings.ads'
    spec of package Greetings

'greetings.adb'
    body of package Greetings

'gmain.adb'
    body of main program
```

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

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

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

```
$ gnatmake gmain.adb
```

## 3 The GNAT Compilation Model

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

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

### 3.1 Source Representation

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

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

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

























































































### 3.11.1 Interfacing to C

Interfacing Ada with a foreign language such as C involves using compiler directives to import and/or export entity definitions in each language – using `extern` statements in C, for example, and the `Import`, `Export`, and `Convention` pragmas in Ada. A full treatment of these topics is provided in Appendix B, section 1 of the Ada Reference Manual.

There are two ways to build a program using GNAT that contains some Ada sources and some foreign language sources, depending on whether or not the main subprogram is written in Ada. Here's an example with the main subprogram in Ada:

```

/* file1.c */
#include <stdio.h>

void print_num (int num)
{
    printf ("num is %d.\n", num);
    return;
}

/* file2.c */

/* num_from_Ada is declared in my_main.adb */
extern int num_from_Ada;

int get_num (void)
{
    return num_from_Ada;
}

-- my_main.adb
procedure My_Main is

    -- Declare then export an Integer entity called num_from_Ada
    My_Num : Integer := 10;
    pragma Export (C, My_Num, "num_from_Ada");

    -- Declare an Ada function spec for Get_Num, then use
    -- C function get_num for the implementation.
    function Get_Num return Integer;
    pragma Import (C, Get_Num, "get_num");

    -- Declare an Ada procedure spec for Print_Num, then use
    -- C function print_num for the implementation.
    procedure Print_Num (Num : Integer);
    pragma Import (C, Print_Num, "print_num");

begin
    Print_Num (Get_Num);
end My_Main;

```

To build this example:

- \* First compile the foreign language files to generate object files:

```
$ gcc -c file1.c
$ gcc -c file2.c
```

- \* Then compile the Ada units to produce a set of object files and ALI files:

```
$ gnatmake -c my_main.adb
```

- \* Run the Ada binder on the Ada main program:

```
$ gnatbind my_main.ali
```

- \* Link the Ada main program, the Ada objects, and the other language objects:

```
$ gnatlink my_main.ali file1.o file2.o
```

You can merge the last three steps into a single command:

```
$ gnatmake my_main.adb -largs file1.o file2.o
```

If the main program is in a language other than Ada, you may have more than one entry point into the Ada subsystem. You must use a special binder option to generate callable routines that initialize and finalize the Ada units ([Binding with Non-Ada Main Programs], page 166). You must insert calls to the initialization and finalization routines in the main program or some other appropriate point in the code. You must place the call to initialize the Ada units so that it occurs before the first Ada subprogram is called and must place the call to finalize the Ada units so it occurs after the last Ada subprogram returns. The binder places the initialization and finalization subprograms into the `b~xxx.adb` file, where they can be accessed by your C sources. To illustrate, we have the following example:

```
/* main.c */
extern void adainit (void);
extern void adafinal (void);
extern int add (int, int);
extern int sub (int, int);

int main (int argc, char *argv[])
{
    int a = 21, b = 7;

    adainit();

    /* Should print "21 + 7 = 28" */
    printf ("%d + %d = %d\n", a, b, add (a, b));

    /* Should print "21 - 7 = 14" */
    printf ("%d - %d = %d\n", a, b, sub (a, b));

    adafinal();
}

-- unit1.ads
package Unit1 is
    function Add (A, B : Integer) return Integer;
```

```

    pragma Export (C, Add, "add");
end Unit1;
-- unit1.adb
package body Unit1 is
    function Add (A, B : Integer) return Integer is
    begin
        return A + B;
    end Add;
end Unit1;
-- unit2.ads
package Unit2 is
    function Sub (A, B : Integer) return Integer;
    pragma Export (C, Sub, "sub");
end Unit2;
-- unit2.adb
package body Unit2 is
    function Sub (A, B : Integer) return Integer is
    begin
        return A - B;
    end Sub;
end Unit2;

```

The build procedure for this application is similar to the last example's:

- \* First, compile the foreign language files to generate object files:
 

```
$ gcc -c main.c
```
- \* Next, compile the Ada units to produce a set of object files and ALI files:
 

```
$ gnatmake -c unit1.adb
$ gnatmake -c unit2.adb
```
- \* Run the Ada binder on every generated ALI file. Make sure to use the `-n` option to specify a foreign main program:
 

```
$ gnatbind -n unit1.ali unit2.ali
```
- \* Link the Ada main program, the Ada objects and the foreign language objects. You need only list the last ALI file here:
 

```
$ gnatlink unit2.ali main.o -o exec_file
```

This procedure yields a binary executable called `exec_file`.

Depending on the circumstances (for example when your non-Ada main object does not provide symbol `main`), you may also need to instruct the GNAT linker not to include the standard startup objects by passing the `-nostartfiles` switch to `gnatlink`.

### 3.11.2 Calling Conventions

GNAT follows standard calling sequence conventions and will 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 is 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. You can also mix GNAT generated code and code generated by another Ada compiler. In this case, you should restrict the data types to simple cases, including primitive types. Whether complex data types can be passed depends on the situation. It is probably 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.

If output from two different compilers is mixed, you are 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 it takes care of its own elaboration requirements, and 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.

#### Assembler

Specifies assembler as the convention. In practice this has the same effect as convention `Ada` (but is not equivalent in the sense of being considered the same convention).

#### Asm

Equivalent to `Assembler`.

#### COBOL

Data is passed according to the conventions described in section B.4 of the Ada Reference Manual.

#### C

Data is passed according to the conventions described in section B.3 of the Ada Reference Manual.

A note on interfacing to a C ‘`varargs`’ function:

In C, `varargs` allows a function to take a variable number of arguments. There is no direct equivalent in this to Ada. One approach that you can use is to create a C wrapper for each different profile and then interface to this C wrapper. For example, to print an `int` value using `printf`, create a C function `printfi` that takes two arguments, a pointer to a string and an `int`, and calls `printf`. Then in the Ada program, use `pragma Import` to interface to `printfi`.

It may work on some platforms to directly interface to a `varargs` function by providing a specific Ada profile for a particular call. However, this does not work on all platforms since there is no guarantee that the calling sequence for a two-argument normal C function is the same as for calling a `varargs` C function with the same two arguments.









```

{
    t->method1 ();
}

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

//ex7.h

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

//ex7.C

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

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

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

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

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

```

```

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

-- simple_cpp_interface.ads
with System;
package Simple_Cpp_Interface is
  type A is limited
    record
      Vptr      : System.Address;
      O_Value   : Integer;
      A_Value   : Integer;
    end record;
  pragma Convention (C, A);

  procedure Method1 (This : in out A);
  pragma Import (C, Method1);

  procedure Ada_Method2 (This : in out A; V : Integer);
  pragma Export (C, Ada_Method2);

end Simple_Cpp_Interface;

-- simple_cpp_interface.adb
package body Simple_Cpp_Interface is

  procedure Ada_Method2 (This : in out A; V : Integer) is
  begin
    Method1 (This);
    This.A_Value := V;
  end Ada_Method2;

end Simple_Cpp_Interface;
```

### 3.11.3.4 Interfacing with C++ constructors

To interface with C++ constructors GNAT provides the `pragma CPP_Constructor` (see the *GNAT\_Reference\_Manual* for additional information). In this section, we present some common uses of C++ constructors in mixed-languages programs in GNAT.

Let us assume we need to interface with the following C++ class:

```

class Root {
public:
  int  a_value;
  int  b_value;
  virtual int Get_Value ();
  Root();           // Default constructor
  Root(int v);      // 1st non-default constructor
  Root(int v, int w); // 2nd non-default constructor
};
```







```

        virtual void Set_Owner (char* Name) = 0;
    };

```

Using these declarations, we can now say that a Dog is an animal that is both Carnivore and Domestic, that is:

```

class Dog : Animal, Carnivore, Domestic {
public:
    virtual int  Number_Of_Teeth ();
    virtual void Set_Owner (char* Name);

    Dog(); // Constructor
private:
    int  Tooth_Count;
    char *Owner;
};

```

In the following examples we assume that the previous declarations are located in a file named `animals.h`. The following package demonstrates how to import these C++ declarations from the Ada side:

```

with Interfaces.C.Strings; use Interfaces.C.Strings;
package Animals is
    type Carnivore is limited interface;
    pragma Convention (C_Plus_Plus, Carnivore);
    function Number_Of_Teeth (X : Carnivore)
        return Natural is abstract;

    type Domestic is limited interface;
    pragma Convention (C_Plus_Plus, Domestic);
    procedure Set_Owner
        (X      : in out Domestic;
         Name : Chars_Ptr) is abstract;

    type Animal is tagged limited record
        Age : Natural;
    end record;
    pragma Import (C_Plus_Plus, Animal);

    procedure Set_Age (X : in out Animal; Age : Integer);
    pragma Import (C_Plus_Plus, Set_Age);

    function Age (X : Animal) return Integer;
    pragma Import (C_Plus_Plus, Age);

    function New_Animal return Animal;
    pragma CPP_Constructor (New_Animal);
    pragma Import (CPP, New_Animal, "_ZN6AnimalC1Ev");

    type Dog is new Animal and Carnivore and Domestic with record

```



```

with Interfaces.C.Strings;
use Interfaces.C.Strings;
package Animals is
  type Carnivore is limited interface;
  pragma Convention (C_Plus_Plus, Carnivore);
  function Number_Of_Teeth (X : Carnivore)
    return Natural is abstract;

  type Domestic is limited interface;
  pragma Convention (C_Plus_Plus, Domestic);
  procedure Set_Owner
    (X      : in out Domestic;
     Name   : Chars_Ptr) is abstract;

  type Animal is tagged record
    Age : Natural;
  end record;
  pragma Convention (C_Plus_Plus, Animal);

  procedure Set_Age (X : in out Animal; Age : Integer);
  pragma Export (C_Plus_Plus, Set_Age);

  function Age (X : Animal) return Integer;
  pragma Export (C_Plus_Plus, Age);

  function New_Animal return Animal'Class;
  pragma Export (C_Plus_Plus, New_Animal);

  type Dog is new Animal and Carnivore and Domestic with record
    Tooth_Count : Natural;
    Owner       : String (1 .. 30);
  end record;
  pragma Convention (C_Plus_Plus, Dog);

  function Number_Of_Teeth (A : Dog) return Natural;
  pragma Export (C_Plus_Plus, Number_Of_Teeth);

  procedure Set_Owner (A : in out Dog; Name : Chars_Ptr);
  pragma Export (C_Plus_Plus, Set_Owner);

  function New_Dog return Dog'Class;
  pragma Export (C_Plus_Plus, New_Dog);
end Animals;

```

Compared with our previous example the only differences are the use of **pragma Convention** (instead of **pragma Import**) and the use of **pragma Export** to indicate to the GNAT compiler that the primitives will be available to C++. Thanks to the ABI compatibility, on the C++

side there is nothing else to be done; as explained above, the only requirement is that all the primitives and components are declared in exactly the same order.

For completeness, let us see a brief C++ main program that uses the declarations available in `animals.h` (presented in our first example) to import and use the declarations from the Ada side, properly initializing and finalizing the Ada run-time system along the way:

```
#include "animals.h"
#include <iostream>
using namespace std;

void Check_Carnivore (Carnivore *obj) {...}
void Check_Domestic (Domestic *obj)  {...}
void Check_Animal (Animal *obj)      {...}
void Check_Dog (Dog *obj)             {...}

extern "C" {
    void adainit (void);
    void adafinal (void);
    Dog* new_dog ();
}

void test ()
{
    Dog *obj = new_dog(); // Ada constructor
    Check_Carnivore (obj); // Check secondary DT
    Check_Domestic (obj); // Check secondary DT
    Check_Animal (obj);   // Check primary DT
    Check_Dog (obj);      // Check primary DT
}

int main ()
{
    adainit (); test(); adafinal ();
    return 0;
}
```

### 3.11.4 Partition-Wide Settings

When building a mixed-language application, you must be aware that Ada enforces some partition-wide settings that may implicitly impact the behavior of the other languages.

This is the case for certain signals that are reserved to the implementation to implement proper Ada semantics (such as the behavior of `abort` statements). It means that the Ada part of the application may override signal handlers that were previously installed by either the system or by other user code.

If your application requires that either system or user signals be preserved, you need to instruct the Ada part not to install its own signal handler. You do this using `pragma Interrupt_State` that provides a general mechanism for overriding such uses of interrupts.





```

public:
    virtual int Number_Of_Teeth () = 0;
};

class Domestic {
public:
    virtual void Set_Owner (char* Name) = 0;
};

class Animal {
public:
    int Age_Count;
    virtual void Set_Age (int New_Age);
};

class Dog : Animal, Carnivore, Domestic {
public:
    int  Tooth_Count;
    char *Owner;

    virtual int  Number_Of_Teeth ();
    virtual void Set_Owner (char* Name);

    Dog();
};

```

The corresponding Ada code is generated:

```

package Class_Carnivore is
    type Carnivore is limited interface;
    pragma Import (CPP, Carnivore);

    function Number_Of_Teeth (this : access Carnivore) return int is abstract;
end;
use Class_Carnivore;

package Class_Domestic is
    type Domestic is limited interface;
    pragma Import (CPP, Domestic);

    procedure Set_Owner
        (this : access Domestic;
         Name : Interfaces.C.Strings.chars_ptr) is abstract;
end;
use Class_Domestic;

package Class_Animal is
    type Animal is tagged limited record

```

```

    Age_Count : aliased int;
end record;
pragma Import (CPP, Animal);

procedure Set_Age (this : access Animal; New_Age : int);
pragma Import (CPP, Set_Age, "_ZN6Animal7Set_AgeEi");
end;
use Class_Animal;

package Class_Dog is
  type Dog is new Animal and Carnivore and Domestic with record
    Tooth_Count : aliased int;
    Owner : Interfaces.C.Strings.chars_ptr;
  end record;
  pragma Import (CPP, Dog);

  function Number_Of_Teeth (this : access Dog) return int;
  pragma Import (CPP, Number_Of_Teeth, "_ZN3Dog15Number_Of_TeethEv");

  procedure Set_Owner
    (this : access Dog; Name : Interfaces.C.Strings.chars_ptr);
  pragma Import (CPP, Set_Owner, "_ZN3Dog9Set_OwnerEPc");

  function New_Dog return Dog;
  pragma CPP_Constructor (New_Dog);
  pragma Import (CPP, New_Dog, "_ZN3DogC1Ev");
end;
use Class_Dog;

```

### 3.11.5.3 Switches

- fdump-ada-spec  
Generate Ada spec files for the given header files transitively (including all header files that these headers depend upon).
- fdump-ada-spec-slim  
Only generate Ada spec files for the header files specified on the command line.
- fada-spec-parent=`unit`  
Specifies that all files generated by -fdump-ada-spec are to be child units of the specified parent unit.
- C  
Extract comments from headers and generate Ada comments in the Ada spec files.

### 3.11.6 Generating C Headers for Ada Specifications

GNAT includes a C header generator for Ada specifications that supports Ada types that have a direct mapping to C types. This specifically includes support for:

- \* Scalar types
- \* Constrained arrays
- \* Records (untagged)
- \* Composition of the above types
- \* Constant declarations
- \* Object declarations
- \* Subprogram declarations

### 3.11.6.1 Running the C Header Generator

The C header generator is part of the GNAT compiler and can be invoked via the `-gnatceg` switch, which generates a `.h` file corresponding to the given input file (Ada spec or body). Note that only spec files are processed, so giving a spec or a body file as input is equivalent. For example:

```
$ gcc -c -gnatceg pack1.ads
```

generates a self-contained file called `pack1.h` including common definitions from the Ada Standard package followed by the definitions included in `pack1.ads` as well as all the other units withed by this file.

For instance, given the following Ada files:

```
package Pack2 is
  type Int is range 1 .. 10;
end Pack2;

with Pack2;

package Pack1 is
  type Rec is record
    Field1, Field2 : Pack2.Int;
  end record;

  Global : Rec := (1, 2);

  procedure Proc1 (R : Rec);
  procedure Proc2 (R : in out Rec);
end Pack1;
```

The above `gcc` command generates the following `pack1.h` file:

```
/* Standard definitions skipped */
#ifndef PACK2_ADS
#define PACK2_ADS
typedef short_short_integer pack2__TintB;
typedef pack2__TintB pack2__int;
#endif /* PACK2_ADS */

#ifndef PACK1_ADS
#define PACK1_ADS
typedef struct _pack1__rec {
```





```
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 you use a pragma `Export`, you may override this:

```
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 whatever link name the C compiler would assign for the C function `var1_name`. This typically would be either `var1_name` or `_var1_name`, depending on operating system conventions, but other possibilities exist. The link name for `Var2` is `var2_link_name`, and this is not operating system dependent.

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, GNAT adds 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 is `_ada_hello`.





















































Don't quit. Generate ALI and tree files even if illegalities. Note that code generation is still suppressed in the presence of any errors, so even with `-gnatQ` no object file is generated.

- `-gnatr`  
Treat pragma Restrictions as Restriction\_Warnings.
- `-gnatR[0|1|2|3|4] [e] [h] [m] [j] [s]`  
Output representation information for declared types, objects and subprograms. Note that this switch is not allowed if a previous `-gnatD` switch has been given, since these two switches are not compatible.
- `-gnats`  
Syntax check only.
- `-gnatS`  
Print package Standard.
- `-gnatT`nnn'`  
All compiler tables start at `nnn` times usual starting size.
- `-gnatu`  
List units for this compilation.
- `-gnatU`  
Tag all error messages with the unique string 'error:'
- `-gnatv`  
Verbose mode. Full error output with source lines to `stdout`.
- `-gnatV`  
Control level of validity checking ([Validity Checking], page 131).
- `-gnatw`xxx'`  
Warning mode where `xxx` is a string of option letters that denotes the exact warnings that are enabled or disabled ([Warning Message Control], page 109).
- `-gnatW`e'`  
Wide character encoding method (`e=n/h/u/s/e/8`).
- `-gnatx`  
Suppress generation of cross-reference information.
- `-gnatX`  
Enable core GNAT implementation extensions and latest Ada version.
- `-gnatX0`  
Enable all GNAT implementation extensions and latest Ada version.
- `-gnaty`  
Enable built-in style checks ([Style Checking], page 135).
- `-gnatz`m'`  
Distribution stub generation and compilation (`m=r/c` for receiver/caller stubs).













- \* Address clauses with possibly unaligned values, or where an attempt is made to overlay a smaller variable with a larger one.
- \* Fixed-point type declarations with a null range
- \* `Direct_IO` or `Sequential_IO` instantiated with a type that has access values
- \* 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` statement along some execution path in a function
- \* Incorrect (unrecognized) pragmas
- \* Incorrect external names
- \* Allocation from empty storage pool
- \* Potentially blocking operation in protected type
- \* Suspicious parenthesization of expressions
- \* Mismatching bounds in an aggregate
- \* Attempt to return local value by reference
- \* Premature instantiation of a generic body
- \* Attempt to pack aliased components
- \* Out of bounds array subscripts
- \* Wrong length on string assignment
- \* Violations of style rules if style checking is enabled
- \* Unused ‘with’ clauses
- \* `Bit_Order` usage that does not have any effect
- \* `Standard.Duration` used to resolve universal fixed expression
- \* Dereference of possibly null value
- \* Declaration that is likely to cause storage error
- \* Internal GNAT unit ‘with’ed by application unit
- \* Values known to be out of range at compile time
- \* Unreferenced or unmodified variables. Note that a special exemption applies to variables which contain any of the substrings `DISCARD`, `DUMMY`, `IGNORE`, `JUNK`, `UNUSE`, `TMP`, `TEMP` in any casing. Such variables are considered likely to be intentionally used in a situation where otherwise a warning would be given, so warnings of this kind are always suppressed for such variables.
- \* Address overlays that could clobber memory
- \* Unexpected initialization when address clause present
- \* Bad alignment for address clause
- \* Useless type conversions





































errors, and prevent the generation of an object file. The use of this switch also sets the default front-end warning mode to `-gnatwe`, that is, front-end warning messages and style check messages are treated as errors as well.

A string of warning parameters can be used in the same parameter. For example:

`-gnatwaGe`

will turn on all optional warnings except for unrecognized pragma warnings, and also specify that warnings should be treated as errors.

When no switch `-gnatw` is used, this is equivalent to:

```
* -gnatw.a
* -gnatwB
* -gnatw.b
* -gnatwC
* -gnatw.C
* -gnatwD
* -gnatw.D
* -gnatwF
* -gnatw.F
* -gnatwg
* -gnatwH
* -gnatw.H
* -gnatwi
* -gnatwJ
* -gnatw.J
* -gnatwK
* -gnatw.K
* -gnatwL
* -gnatw.L
* -gnatwM
* -gnatw.m
* -gnatwn
* -gnatw.N
* -gnatwo
* -gnatw.O
* -gnatwP
* -gnatw.P
* -gnatwq
* -gnatw.Q
* -gnatwR
* -gnatw.R
```



























































































































You can't refer to the original generic entities in GDB, but you can 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:

```
(gdb) 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 you do for other units.

### 6.1.10 Remote Debugging with gdbserver

On platforms that support **gdbserver**, you can use this tool to debug your application remotely. This can be useful in situations where the program needs to be run on a target host that is different from the host used for development, particularly when the target has a limited amount of resources (either CPU and/or memory).

To do so, start your program using **gdbserver** on the target machine. **gdbserver** automatically suspends the execution of your program at its entry point, waiting for a debugger to connect to it. You use the following commands to start an application and tell **gdbserver** to wait for a connection with the debugger on localhost port 4444.

```

$ gdbserver localhost:4444 program
Process program created; pid = 5685
Listening on port 4444

```

Once `gdbserver` has started listening, you can tell the debugger to establish a connection with this `gdbserver`, and then start a debugging session as if the program was being debugged on the same host, directly under the control of GDB.

```
$ gdb program
(gdb) target remote targethost:4444
Remote debugging using targethost:4444
0x00007f29936d0af0 in ?? () from /lib64/ld-linux-x86-64.so.
(gdb) b foo.adb:3
Breakpoint 1 at 0x401f0c: file foo.adb, line 3.
(gdb) continue
Continuing.

Breakpoint 1, foo () at foo.adb:4
4         end foo;
```

You can also use `gdbserver` to attach to an already running program, in which case the execution of that program is suspended until you have established the connection between the debugger and `gdbserver`.

For more information on how to use `gdbserver`, see the ‘Using the `gdbserver` Program’ section in *Debugging with GDB*. GNAT provides support for `gdbserver` on x86-linux, x86-windows and x86\_64-linux.

### 6.1.11 GNAT Abnormal Termination or Failure to Terminate

When presented with programs that contain serious errors in syntax or semantics, GNAT may, on rare occasions, experience problems such as aborting with a segmentation fault or illegal memory access, raising an internal exception, terminating abnormally, or failing to terminate at all. 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 for you to use in such cases are presented in increasing order of difficulty, corresponding to your experience in using GNAT and your familiarity with compiler internals.

- \* Run `gcc` with the `-gnatf`. This switch causes all errors on a given line to be reported. In its absence, GNAT only displays the first error on a line.  
The `-gnatd0` switch causes errors to be displayed as soon as they are encountered, rather than after compilation is terminated. If GNAT terminates prematurely or goes into an infinite loop, the last error message displayed may help to pinpoint the culprit.
- \* 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 problem.
- \* Run `gcc` with the `-gnatdc` switch. This is a GNAT specific switch that does for the front-end what `-v` does for the back end. The system prints the name of each unit, either a compilation unit or nested unit, as it is being analyzed.
- \* 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 [The GNAT Debugger GDB],

page 184, 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** back end, indicates the source line at which the execution stopped, and **input\_file name** indicates the name of the source file.

### 6.1.12 Naming Conventions for GNAT Source Files

In order to better understand 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 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, such as **sem\_aggr.adb** for aggregates and **sem\_disp.adb** for dynamic dispatching.
- \* All files prefixed with **exp** perform normalization and expansion of the intermediate representation (abstract syntax tree, or AST). The expansion has the effect of lowering the semantic level of the AST to a level closer to what the back end can handle. For example, it converts tasking operations into calls to the appropriate runtime routines. 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.
- \* The files prefixed with **gen\_il** generate most of the functions defined in **sinfo.ads** and **einfo.ads**, which set and get various fields and flags of the AST.
- \* 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 specs. All the other **.c** files are modifications of common **gcc** files.

### 6.1.13 Getting Internal Debugging Information

Most compilers have internal debugging switches and modes. GNAT does too, 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 shown explicitly, complex aggregates are replaced with loops and assignments, and tasking primitives are replaced with run-time calls.

### 6.1.14 Stack Traceback

Traceback is a mechanism to display the sequence of subprogram calls that leads to a specified execution point in a program. Often (but not always) the execution point is an instruction at which an exception has been raised. This mechanism is also known as ‘stack unwinding’ because it obtains its information by scanning the run-time stack and recovering the activation records of all active subprograms. Stack unwinding is one of the most important tools for program debugging.

The first entry stored in traceback corresponds to the deepest calling level, that is to say the subprogram currently executing the instruction from which we want to obtain the traceback.

Note that there is no runtime performance penalty when stack traceback is enabled and no exception is raised during program execution.

#### 6.1.14.1 Non-Symbolic Traceback

Note: this feature is not supported on all platforms. See `GNAT.Traceback` spec in `g-traceb.ads` for a complete list of supported platforms.

### Tracebacks From an Unhandled Exception

A runtime non-symbolic traceback is a list of addresses of call instructions. To enable this feature you must use the `-E gnatbind` switch. With this switch, a stack traceback is stored at runtime as part of exception information.

You can translate this information using the `addr2line` tool, provided that the program is compiled with debugging options (see [Compiler Switches], page 90) and linked at a fixed position with `-no-pie`.

Here’s a simple example with `gnatmake`:

```

procedure STB is

  procedure P1 is
  begin
    raise Constraint_Error;
  end P1;

  procedure P2 is

```

```

begin
  P1;
end P2;

begin
  P2;
end STB;

$ gnatmake stb -g -bargs -E -largs -no-pie
$ stb

Execution of stb terminated by unhandled exception
raised CONSTRAINT_ERROR : stb.adb:5 explicit raise
Load address: 0x400000
Call stack traceback locations:
0x401373 0x40138b 0x40139c 0x401335 0x4011c4 0x4011f1 0x77e892a4

```

As we can see, the traceback lists a sequence of addresses for the unhandled exception `CONSTRAINT_ERROR` raised in procedure `P1`. It's easy to see that this exception come from procedure `P1`. To translate these addresses into the source lines where the calls appear, you need to invoke the `addr2line` tool like this:

```

$ addr2line -e stb 0x401373 0x40138b 0x40139c 0x401335 0x4011c4
0x4011f1 0x77e892a4

d:/stb/stb.adb:5
d:/stb/stb.adb:10
d:/stb/stb.adb:14
d:/stb/b~stb.adb:197
crtexe.c:?
crtexe.c:?
??:0

```

The `addr2line` tool has several other useful options:

<code>-a --addresses</code>	to show the addresses alongside the line numbers
<code>-f --functions</code>	to get the function name corresponding to a location
<code>-p --pretty-print</code>	to print all the information on a single line
<code>--demangle=gnat</code>	to use the GNAT decoding mode for the function names

```

$ addr2line -e stb -a -f -p --demangle=gnat 0x401373 0x40138b
0x40139c 0x401335 0x4011c4 0x4011f1 0x77e892a4

0x00401373: stb.p1 at d:/stb/stb.adb:5
0x0040138B: stb.p2 at d:/stb/stb.adb:10
0x0040139C: stb at d:/stb/stb.adb:14
0x00401335: main at d:/stb/b~stb.adb:197

```



```
0x004011f1 mainCRTStartup at ???
0x77e892a4 ??? at ???
```

## Tracebacks From Exception Occurrences

Non-symbolic tracebacks are obtained by using the `-E` binder switch. The stack traceback is attached to the exception information string and you can retrieve it in an exception handler within the Ada program by means of the Ada facilities defined in `Ada.Exceptions`. Here's a simple example:

```
with Ada.Text_IO;
with Ada.Exceptions;

procedure STB is

  use Ada;
  use Ada.Exceptions;

  procedure P1 is
    K : Positive := 1;
  begin
    K := K - 1;
  exception
    when E : others =>
      Text_IO.Put_Line (Exception_Information (E));
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  P2;
end STB;

$ gnatmake stb -g -bargs -E -largh -no-pie
$ stb

raised CONSTRAINT_ERROR : stb.adb:12 range check failed
Load address: 0x400000
Call stack traceback locations:
0x4015e4 0x401633 0x401644 0x401461 0x4011c4 0x4011f1 0x77e892a4
```

## Tracebacks From Anywhere in a Program

You can also retrieve a stack traceback from anywhere in a program. For this, you need to use the `GNAT.Traceback` API. This package includes a procedure called `Call_Chain` that computes a complete stack traceback as well as useful display procedures described below.

You don't have to use the `-E gnatbind` switch in this case because the stack traceback mechanism is invoked explicitly.

In the following example, we compute a traceback at a specific location in the program and display it using `GNAT.Debug_Uilities.Image` to convert addresses to strings:

```

with Ada.Text_IO;
with GNAT.Traceback;
with GNAT.Debug_Uilities;
with System;

procedure STB is

  use Ada;
  use Ada.Text_IO;
  use GNAT;
  use GNAT.Traceback;
  use System;

  LA : constant Address := Executable_Load_Address;

  procedure P1 is
    TB : Tracebacks_Array (1 .. 10);
    -- We are asking for a maximum of 10 stack frames.
    Len : Natural;
    -- Len will receive the actual number of stack frames returned.
  begin
    Call_Chain (TB, Len);

    Put ("In STB.P1 : ");

    for K in 1 .. Len loop
      Put (Debug_Uilities.Image_C (TB (K)));
      Put (' ');
    end loop;

    New_Line;
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  if LA /= Null_Address then
    Put_Line ("Load address: " & Debug_Uilities.Image_C (LA));
  end if;

```

```

        P2;
    end STB;

$ gnatmake stb -g
$ stb

Load address: 0x400000
In STB.P1 : 0x40F1E4 0x4014F2 0x40170B 0x40171C 0x401461 0x4011C4 \
0x4011F1 0x77E892A4

```

You can get even more information by invoking the `addr2line` tool or the `gnatsymbolize` tool as described earlier (note that the hexadecimal addresses need to be specified in C format, with a leading ‘0x’).

### 6.1.14.2 Symbolic Traceback

A symbolic traceback is a stack traceback in which procedure names are associated with each code location.

Note that this feature is not supported on all platforms. See `GNAT.Traceback.Symbolic` spec in `g-trasym.ads` for a complete list of currently supported platforms.

Note that the symbolic traceback requires that the program be compiled with debug information. If you do not compile it with debug information, only the non-symbolic information will be valid.

## Tracebacks From Exception Occurrences

Here is an example:

```

with Ada.Text_IO;
with GNAT.Traceback.Symbolic;

procedure STB is

    procedure P1 is
    begin
        raise Constraint_Error;
    end P1;

    procedure P2 is
    begin
        P1;
    end P2;

    procedure P3 is
    begin
        P2;
    end P3;

begin

```

```

        P3;
exception
  when E : others =>
    Ada.Text_IO.Put_Line (GNAT.Traceback.Symbolic.Symbolic_Traceback (E));
end STB;

$ gnatmake -g stb -bargs -E
$ stb

0040149F in stb.p1 at stb.adb:8
004014B7 in stb.p2 at stb.adb:13
004014CF in stb.p3 at stb.adb:18
004015DD in ada.stb at stb.adb:22
00401461 in main at b~stb.adb:168
004011C4 in __mingw_CRTStartup at crt1.c:200
004011F1 in mainCRTStartup at crt1.c:222
77E892A4 in ?? at ??:0

```

## Tracebacks From Anywhere in a Program

You can get a symbolic stack traceback from anywhere in a program, just as you can for non-symbolic tracebacks. The first step is to obtain a non-symbolic traceback. Then call `Symbolic_Traceback` to compute the symbolic information. Here is an example:

```

with Ada.Text_IO;
with GNAT.Traceback;
with GNAT.Traceback.Symbolic;

procedure STB is

  use Ada;
  use GNAT.Traceback;
  use GNAT.Traceback.Symbolic;

  procedure P1 is
    TB : Tracebacks_Array (1 .. 10);
    -- We are asking for a maximum of 10 stack frames.
    Len : Natural;
    -- Len will receive the actual number of stack frames returned.
  begin
    Call_Chain (TB, Len);
    Text_IO.Put_Line (Symbolic_Traceback (TB (1 .. Len)));
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

```

```
begin
  P2;
end STB;
```

## Automatic Symbolic Tracebacks

You may also enable symbolic tracebacks by using the `-Es` switch to `gnatbind` (as in `gprbuild -g ... -bargs -Es`). This causes the `Exception.Information` to contain a symbolic traceback, which will also be printed if an unhandled exception terminates the program.

### 6.1.15 Pretty-Printers for the GNAT runtime

As discussed in *Calling User-Defined Subprograms*, GDB's `print` command only knows about the physical layout of program data structures and therefore normally displays only low-level dumps, which are often hard to understand.

An example of this is when trying to display the contents of an Ada standard container, such as `Ada.Containers.Ordered_Maps.Map`:

```
with Ada.Containers.Ordered_Maps;

procedure PP is
  package Int_To_Nat is
    new Ada.Containers.Ordered_Maps (Integer, Natural);

    Map : Int_To_Nat.Map;
begin
  Map.Insert (1, 10);
  Map.Insert (2, 20);
  Map.Insert (3, 30);

  Map.Clear; -- BREAK HERE
end PP;
```

When this program is built with debugging information and run under GDB up to the `Map.Clear` statement, trying to print `Map` will yield information that is only relevant to the developers of the standard containers:

```
(gdb) print map
$1 = (
  tree => (
    first => 0x64e010,
    last => 0x64e070,
    root => 0x64e040,
    length => 3,
    tc => (
      busy => 0,
      lock => 0
    )
  )
)
```

Fortunately, GDB has a feature called `pretty-printers` [http://docs.adacore.com/gdb-docs/html/gdb.html#Pretty\\_002dPrinter-Introduction](http://docs.adacore.com/gdb-docs/html/gdb.html#Pretty_002dPrinter-Introduction), which allows customizing how GDB displays data structures. The GDB shipped with GNAT embeds such pretty-printers for the most common containers in the standard library. To enable them, either run the following command manually under GDB or add it to your `.gdbinit` file:

```
python import gnatdbg; gnatdbg.setup()
```

Once you've done this, GDB's `print` command will automatically use these pretty-printers when appropriate. Using the previous example:

```
(gdb) print map
$1 = pp.int_to_nat.map of length 3 = {
  [1] = 10,
  [2] = 20,
  [3] = 30
}
```

Pretty-printers are invoked each time GDB tries to display a value, including when displaying the arguments of a called subprogram (in GDB's `backtrace` command) or when printing the value returned by a function (in GDB's `finish` command).

To display a value without involving pretty-printers, you can invoke `print` with its `/r` option:

```
(gdb) print/r map
$1 = (
  tree => (...)
```

You can also obtain finer control of pretty-printers: see GDB's online documentation<sup>1</sup> for more information.

## 6.2 Profiling

This section describes how to use the `gprof` profiler tool on Ada programs.

### 6.2.1 Profiling an Ada Program with `gprof`

This section is not meant to be an exhaustive documentation of `gprof`. You can find full documentation for it in the *GNU Profiler User's Guide* documentation that is part of this GNAT distribution.

Profiling a program helps determine the parts of a program that are executed most often and are therefore the most time-consuming.

`gprof` is the standard GNU profiling tool; it has been enhanced to better handle Ada programs and multitasking. It's currently supported on the following platforms

- \* Linux x86/x86\_64
- \* Windows x86/x86\_64 (without PIE support)

In order to profile a program using `gprof`, you need to perform the following steps:

1. Instrument the code, which requires a full recompilation of the project with the proper switches.

<sup>1</sup> [http://docs.adacore.com/gdb-docs/html/gdb.html#Pretty\\_002dPrinter-Commands](http://docs.adacore.com/gdb-docs/html/gdb.html#Pretty_002dPrinter-Commands)

2. Execute the program under the analysis conditions, i.e. with the desired input.
3. Analyze the results using the **gprof** tool.

The following sections detail the different steps and indicate how to interpret the results.

### 6.2.1.1 Compilation for profiling

In order to profile a program, you must first tell the compiler to generate the necessary profiling information. You do this using the compiler switch **-pg**, which you must add to other compilation switches. You need to specify this switch during compilation and link stages, and you can specify it only once when using **gnatmake**:

```
$ gnatmake -f -pg -P my_project
```

Note that only the objects that were compiled with the **-pg** switch will be profiled; if you need to profile your whole project, use the **-f gnatmake** switch to force full recompilation.

Note that on Windows, **gprof** does not support PIE. You should add the **-no-pie** switch to the linker flags to disable PIE.

### 6.2.1.2 Program execution

Once the program has been compiled for profiling, you can run it as usual.

The only constraint imposed by profiling is that the program must terminate normally. An interrupted program (via a Ctrl-C, kill, etc.) will not be properly analyzed.

Once the program completes execution, a data file called **gmon.out** is generated in the directory where the program was launched from. If this file already exists, it will be overwritten by running the program.

### 6.2.1.3 Running gprof

You can call the **gprof** tool as follows:

```
$ gprof my_prog gmon.out
```

or simply:

```
$ gprof my_prog
```

The complete form of the **gprof** command line is the following:

```
$ gprof [switches] [executable [data-file]]
```

**gprof** supports numerous switches, whose order does not matter. You can find the full list of switches in the *GNU Profiler User's Guide*.

The following are the most relevant of those switches:

**--demangle[=*style*], --no-demangle**

These switches control whether symbol names should be demangled when printing output. The default is to demangle C++ symbols. You can use **--no-demangle** to turn off demangling. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler, in particular Ada symbols generated by GNAT can be demangled using **--demangle=gnat**.

**-e *function\_name***

The **-e *function\_name*** option tells **gprof** not to print information about the function *function\_name* and its children in the call graph. The function will still

be listed as a child of any functions that call it, but its index number will be shown as [not printed]. You may specify more than one `-e` switch, but you may only include one `function_name` with each `-e` switch.

**`-E `function_name``**

The `-E `function`` switch works like the `-e` switch, but execution time spent in the function (and children who were not called from anywhere else) will not be used to compute the percentages-of-time for the call graph. You may specify more than one `-E` switch, but you may only include one `function_name` with each `-E` switch.

**`-f `function_name``**

The `-f `function`` switch causes `gprof` to limit the call graph to the function `function_name` and its children and their children. You may specify more than one `-f` switch, but you may only include one `function_name` with each `-f` switch.

**`-F `function_name``**

The `-F `function`` switch works like the `-f` switch, but only time spent in the function and its children and their children will be used to determine total-time and percentages-of-time for the call graph. You may specify more than one `-F` switch, but you may include only one `function_name` with each `-F` switch. The `-F` switch overrides the `-E` switch.

#### 6.2.1.4 Interpretation of profiling results

The results of the profiling analysis are represented by two arrays: the ‘flat profile’ and the ‘call graph’. You can find full documentation of those outputs in the *GNU Profiler User’s Guide*.

The flat profile shows the time spent in each function of the program and how many times it has been called. This allows you to easily locate the most time-consuming functions.

The call graph shows, for each subprogram, the subprograms that call it, and the subprograms that it calls. It also provides an estimate of the time spent in each of those callers and called subprograms.

## 6.3 Improving Performance

This section presents several topics related to program performance. It first describes some of the tradeoffs that you need to consider and some of the techniques for making your program run faster.

It then documents the unused subprogram/data elimination feature, which can reduce the size of program executables.

### 6.3.1 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 default (if you don't select any switches) aims at improving the speed of compilation and minimizing dependences, at the expense of performance of the generated code and consists of:

- \* 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 section describes how you can modify these choices and also provides some guidelines on debugging optimized code.

### 6.3.1.1 Controlling Run-Time Checks

By default, GNAT generates all run-time checks, except stack overflow checks and checks for access before elaboration on subprogram calls. The latter are not required in default mode because all necessary checking is done at compile time.

The GNAT switch, `-gnatp` allows you to modify this default; see [Run-Time Checks], page 142.

Our experience is that the default is suitable for most development purposes.

Elaboration checks are off by default and also not needed by default since GNAT uses a static elaboration analysis approach that avoids the need for run-time checking. This manual contains a full chapter discussing the issue of elaboration checks and you should read this chapter if the default is not satisfactory for your use,

For validity checks, the minimal checks required by the Ada Reference Manual (for case statements and assignments to array elements) are enabled by default. You can suppress these by using the `-gnatVn` switch. Note that in Ada 83, there were no validity checks, so if the Ada 83 mode is acceptable (or when comparing GNAT performance with an Ada 83 compiler), it may be reasonable to routinely use `-gnatVn`. Validity checks are also suppressed entirely if you use `-gnatp`.

Note that the setting of the switches controls the default setting of the checks. You may modify them using either `pragma Suppress` (to remove checks) or `pragma Unsuppress` (to add back suppressed checks) in your program source.

### 6.3.1.2 Use of Restrictions

You can use `pragma Restrictions` to control which features are permitted in your program. In most cases, the use of this pragma itself does not affect the generated code (but, of course, if you avoid relatively expensive features like finalization, you'll have more efficient programs and that's enforceable by the use of `pragma Restrictions (No_Finalization)`).

One notable exception to this rule is that the possibility of task abort results in some distributed overhead, particularly if finalization or exception handlers are used. This is because certain sections of code must be marked as non-abortable.

If you use neither the `abort` statement nor asynchronous transfer of control (`select ... then abort`), this distributed overhead can be removed, which may have a general positive effect in improving overall performance, especially in code involving frequent use of tasking

constructs and controlled types, which will show much improved performance. The relevant restrictions pragmas are

```
pragma Restrictions (No_Abort_Statements);
pragma Restrictions (Max_Asynchronous_Select_Nesting => 0);
```

We recommend that you use these restriction pragmas if possible. If you do this, it also means you can write code without worrying about the possibility of an immediate abort at any point.

### 6.3.1.3 Optimization Levels

Without any optimization switch, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. This means that statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the subprogram and get exactly the results you would expect from the source code. However, the generated programs are considerably larger and slower than when optimization is enabled.

Turning on optimization makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

You can pass the `-O` switch, with or without an operand (the permitted forms with an operand are `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`, and `-Og`) to `gcc` to control the optimization level. If you pass multiple `-O` switches, with or without an operand, the last such switch is the one that's used:

\*

`-O0`

No optimization (the default); generates unoptimized code but has the fastest compilation time. Debugging is easiest with this switch.

Note that many other compilers do substantial optimization even if 'no optimization' is specified. With GCC, it is very unusual to use `-O0` for production if execution time is of any concern, since `-O0` means (almost) no optimization. You should keep this difference between GCC and other compilers in mind when doing performance comparisons.

\*

`-O1`

Moderate optimization (same as `-O` without an operand); optimizes reasonably well but does not degrade compilation time significantly. You may not be able to see some variables in the debugger, and changing the value of some variables in the debugger may not have the effect you desire.

\*

`-O2`

Extensive optimization; generates highly optimized code but has an increased compilation time. You may see significant impacts on your ability to display and modify variables in the debugger.

- \*  
-O3  
Full optimization; attempts more sophisticated transformations, in particular on loops, possibly at the cost of larger generated code. You may be hardly able to use the debugger at this optimization level.
- \*  
-Os  
Optimize for size (code and data) of resulting binary rather than speed; based on the -O2 optimization level, but disables some of its transformations that often increase code size, as well as performs further optimizations designed to reduce code size.
- \*  
-Oz  
Optimize aggressively for size (code and data) of resulting binary rather than speed; may increase the number of instructions executed if these instructions require fewer bytes to be encoded.
- \*  
-Og  
Optimize for debugging experience rather than speed; based on the -O1 optimization level, but attempts to eliminate all the negative effects of optimization on debugging.

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.

Since the precise set of optimizations done at each level will vary from release to release (and sometime from target to target), it is best to think of the optimization settings in general terms. See the ‘Options That Control Optimization’ section in *Using the GNU Compiler Collection (GCC)* for details about the -O settings and a number of -f switches that individually enable or disable specific optimizations.

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.

A note regarding the use of -O3: The use of this optimization level ought not to be automatically preferred over that of level -O2, since it often results in larger executables which may run more slowly. See further discussion of this point in [Inlining of Subprograms], page 209.

#### 6.3.1.4 Debugging Optimized Code

Although it is possible to do a reasonable amount of debugging at nonzero optimization levels, the higher the level the more likely that source-level constructs will have been elimi-

nated by optimization. For example, if a loop is strength-reduced, the loop control variable may be completely eliminated and thus cannot be displayed in the debugger. This can only happen at `-O2` or `-O3`. Explicit temporary variables that you code might be eliminated at level `-O1` or higher.

The use of the `-g` switch, which is needed for source-level debugging, affects the size of the program executable on disk, and indeed the debugging information can be quite large. However, it has no effect on the generated code (and thus does not degrade performance).

Since the compiler generates debugging tables for a compilation unit before it performs optimizations, the optimizing transformations may invalidate some of the debugging data. You therefore need to anticipate certain anomalous situations that may arise while debugging optimized code. These are the most common cases:

- \* ‘The ‘hopping Program Counter’:’ Repeated `step` or `next` commands show the PC bouncing back and forth in the code. This may result from any of the following optimizations:
  - ‘Common subexpression elimination:’ using a single instance of code for a quantity that the source computes several times. As a result you may not be able to stop on what looks like a statement.
  - ‘Invariant code motion:’ moving an expression that does not change within a loop to the beginning of the loop.
  - ‘Instruction scheduling:’ moving instructions so as to overlap loads and stores (typically) with other code or in general to move computations of values closer to their uses. Often this causes you to pass an assignment statement without the assignment happening and then later bounce back to the statement when the value is actually needed. Placing a breakpoint on a line of code and then stepping over it may, therefore, not always cause all the expected side-effects.
- \* ‘The ‘big leap’:’ More commonly known as ‘cross-jumping’, in which two identical pieces of code are merged and the program counter suddenly jumps to a statement that is not supposed to be executed, simply because it (and the code following) translates to the same thing as the code that ‘was’ supposed to be executed. This effect is typically seen in sequences that end in a jump, such as a `goto`, a `return`, or a `break` in a C `switch` statement.
- \* ‘The ‘roving variable’:’ The symptom is an unexpected value in a variable. There are various reasons for this effect:
  - In a subprogram prologue, a parameter may not yet have been moved to its ‘home’.
  - A variable may be dead and its register re-used. This is probably the most common cause.
  - As mentioned above, the assignment of a value to a variable may have been moved.
  - A variable may be eliminated entirely by value propagation or other means. In this case, GCC may incorrectly generate debugging information for the variable.

In general, when an unexpected value appears for a local variable or parameter you should first ascertain if that value was actually computed by your program as opposed to being incorrectly reported by the debugger. Record fields or array elements in an object designated by an access value are generally less of a problem once you have verified that the access value is sensible. Typically, this means checking variables in

the preceding code and in the calling subprogram to verify that the value observed is explainable from other values (you must apply the procedure recursively to those other values); or re-running the code and stopping a little earlier (perhaps before the call) and stepping to better see how the variable obtained the value in question; or continuing to step ‘from’ the point of the strange value to see if code motion had simply moved the variable’s assignments later.

In light of such anomalies, a recommended technique is to use `-O0` early in the software development cycle, when extensive debugging capabilities are most needed, and then move to `-O1` and later `-O2` as the debugger becomes less critical. Whether to use the `-g` switch in the release version is a release management issue. Note that if you use `-g` you can then use the `strip` program on the resulting executable, which removes both debugging information and global symbols.

### 6.3.1.5 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 something that the back end cannot support in inlined subprograms.
- \* Any one of the following applies: `pragma Inline` is applied to the subprogram; the subprogram is local to the unit and called once from within it; the subprogram is small and optimization level `-O2` is specified; optimization level `-O3` is specified; or the subprogram is an expression function.

Calls to subprograms in ‘with’ed units are normally not inlined. To achieve inlining in those case (that is, replacement of the call by the code in the body of the subprogram), 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 something that the back end cannot support in inlined subprograms.
- \* There is a `pragma Inline` for the subprogram.
- \* The `-gnatn` switch is used on the command line.

Even if all these conditions are met, it may not be possible for the compiler to inline the call due to the length of the body, or features in the body that make it impossible for the compiler to do the inlining.

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 the back end.

The use of front end inlining with `-gnatN` generates similar additional dependencies.

Note: The `-fno-inline` switch overrides all other conditions and ensures that no inlining occurs, unless requested with pragma `Inline_Always` for most back ends. The extra dependencies resulting from `-gnatn` will still be active, even if this switch is used to suppress the resulting inlining actions.

For the GCC back end, you can use the `-fno-inline-functions` switch to prevent automatic inlining of subprograms if you use `-O3`.

For the GCC back end, you can use the `-fno-inline-small-functions` switch to prevent automatic inlining of small subprograms if you use `-O2`.

For the GC back end, you can use the `-fno-inline-functions-called-once` switch to prevent inlining of subprograms local to the unit and called once from within it if you use `-O1`.

A note regarding the use of `-O3`: `-gnatn` is made up of two sub-switches `-gnatn1` and `-gnatn2` that you can directly specify. `-gnatn` is translated into one of them based on the optimization level. With `-O2` or below, `-gnatn` is equivalent to `-gnatn1` which activates pragma `Inline` with moderate inlining across modules. With `-O3`, `-gnatn` is equivalent to `-gnatn2` which activates pragma `Inline` with full inlining across modules. If you have used pragma `Inline` in appropriate cases, it's usually much better to use `-O2` and `-gnatn` and avoid the use of `-O3` which has the additional effect of inlining subprograms you did not think should be inlined. We have found that the use of `-O3` may slow down the compilation and increase the code size by performing excessive inlining, leading to increased instruction cache pressure from the increased code size and thus minor performance degradations. So the bottom line here is that you should not automatically assume that `-O3` is better than `-O2` and indeed you should use `-O3` only if tests show that it actually improves performance for your program.

### 6.3.1.6 Floating Point Operations

On almost all targets, GNAT maps `Float` and `Long_Float` to the 32-bit and 64-bit standard IEEE floating-point representations and operations will use standard IEEE arithmetic as provided by the processor. On most, but not all, architectures, the attribute `Machine_Overflows` is `False` for these types, meaning that the semantics of overflow is

implementation-defined. In the case of GNAT, these semantics correspond to the normal IEEE treatment of infinities and NaN (not a number) values. For example,  $1.0 / 0.0$  yields plus infinity and  $0.0 / 0.0$  yields a NaN. By avoiding explicit overflow checks, the performance is greatly improved on many targets. However, if required, you can enable floating-point overflow by using the pragma `Check_Float_Overflow`.

Another consideration that applies specifically to x86 32-bit architectures is which form of floating-point arithmetic is used. By default, the operations use the old style x86 floating-point, which implements an 80-bit extended precision form (on these architectures the type `Long_Long_Float` corresponds to that form). In addition, generation of efficient code in this mode means that the extended precision form is used for intermediate results. This may be helpful in improving the final precision of a complex expression, but it means that the results obtained on the x86 may be different from those on other architectures and, for some algorithms, the extra intermediate precision can be detrimental.

In addition to this old-style floating-point, all modern x86 chips implement an alternative floating-point operation model referred to as SSE2. In this model, there is no extended form and execution performance is significantly enhanced. To force GNAT to use this more modern form, use both of the switches:

```
-msse2 -mfpmath=sse
```

A unit compiled with these switches will automatically use the more efficient SSE2 instruction set for `Float` and `Long_Float` operations. Note that the ABI has the same form for both floating-point models, so you can mix units compiled with and without these switches.

### 6.3.1.7 Vectorization of loops

The GCC and LLVM back ends have an auto-vectorizer that's enabled by default at some optimization levels. For the GCC back end, it's enabled by default at `-O3` and you can request it at other levels with `-ftree-vectorize`. For the LLVM back end, it's enabled by default at lower levels, but you can explicitly enable or disable it with the `-fno-vectorize`, `-fvectorize`, `-fno-slp-vectorize`, and `-fslp-vectorize` switches.

To get auto-vectorization, you also need to make sure that the target architecture features a supported SIMD instruction set. For example, for the x86 architecture, you should at least specify `-msse2` to get significant vectorization (but you don't need to specify it for x86-64 as it is part of the base 64-bit architecture). Similarly, for the PowerPC architecture, you should specify `-maltivec`.

The preferred loop form for vectorization is the `for` iteration scheme. Loops with a `while` iteration scheme can also be vectorized if they are very simple, but the vectorizer will quickly give up otherwise. With either iteration scheme, the flow of control must be straight, in particular no `exit` statement may appear in the loop body. The loop may however contain a single nested loop, if it can be vectorized when considered alone:

```
A : array (1..4, 1..4) of Long_Float;
S : array (1..4) of Long_Float;

procedure Sum is
begin
  for I in A'Range(1) loop
    for J in A'Range(2) loop
```

```

        S (I) := S (I) + A (I, J);
    end loop;
end loop;
end Sum;

```

The vectorizable operations depend on the targeted SIMD instruction set, but addition and some multiplication operators are generally supported, as well as the logical operators for modular types. Note that compiling with `-gnatp` might well reveal cases where some checks do thwart vectorization.

Type conversions may also prevent vectorization if they involve semantics that are not directly supported by the code generator or the SIMD instruction set. A typical example is direct conversion from floating-point to integer types. The solution in this case is to use the following idiom:

```
Integer (S'Truncation (F))
```

if `S` is the subtype of floating-point object `F`.

In most cases, the vectorizable loops are loops that iterate over arrays. All kinds of array types are supported, i.e. constrained array types with static bounds:

```
type Array_Type is array (1 .. 4) of Long_Float;
```

constrained array types with dynamic bounds:

```
type Array_Type is array (1 .. Q.N) of Long_Float;
```

```
type Array_Type is array (Q.K .. 4) of Long_Float;
```

```
type Array_Type is array (Q.K .. Q.N) of Long_Float;
```

or unconstrained array types:

```
type Array_Type is array (Positive range <>) of Long_Float;
```

The quality of the generated code decreases when the dynamic aspect of the array type increases, the worst code being generated for unconstrained array types. This is because the less information the compiler has about the bounds of the array, the more fallback code it needs to generate in order to fix things up at run time.

You can specify that a given loop should be subject to vectorization preferably to other optimizations by means of pragma `Loop_Optimize`:

```
pragma Loop_Optimize (Vector);
```

placed immediately within the loop will convey the appropriate hint to the compiler for this loop. This is currently only supported for the GCC back end.

You can also help the compiler generate better vectorized code for a given loop by asserting that there are no loop-carried dependencies in the loop. Consider for example the procedure:

```
type Arr is array (1 .. 4) of Long_Float;
```

```

procedure Add (X, Y : not null access Arr; R : not null access Arr) is
begin
  for I in Arr'Range loop
    R(I) := X(I) + Y(I);
  end loop;

```

```
end;
```

By default, the compiler cannot unconditionally vectorize the loop because assigning to a component of the array designated by *R* in one iteration could change the value read from the components of the array designated by *X* or *Y* in a later iteration. As a result, the compiler will generate two versions of the loop in the object code, one vectorized and the other not vectorized, as well as a test to select the appropriate version at run time. This can be overcome by another hint:

```
pragma Loop_Optimize (Ivdep);
```

placed immediately within the loop will tell the compiler that it can safely omit the non-vectorized version of the loop as well as the run-time test. This is also currently only supported by the GCC back end.

### 6.3.1.8 Other Optimization Switches

You can also use any specialized optimization switches supported by the back end being used. These switches have not been extensively tested with GNAT but can generally be expected to work. Examples of switches in this category for the GCC back end are `-funroll-loops` and the various target-specific `-m` options (in particular, it has been observed that `-march=xxx` can significantly improve performance on appropriate machines). For full details of these switches, see the ‘Submodel Options’ section in the ‘Hardware Models and Configurations’ chapter of *Using the GNU Compiler Collection (GCC)*.

### 6.3.1.9 Optimization and Strict Aliasing

The strong typing capabilities of Ada allow an optimizer to generate efficient code in situations where other languages would be forced to make worst case assumptions preventing such optimizations. Consider the following example:

```
procedure M is
  type Int1 is new Integer;
  I1 : Int1;

  type Int2 is new Integer;
  type A2 is access Int2;
  V2 : A2;
  ...

begin
  ...
  for J in Data'Range loop
    if Data (J) = I1 then
      V2.all := V2.all + 1;
    end if;
  end loop;
  ...
end;
```

Here, since *V2* can only access objects of type *Int2* and *I1* is not one of them, there is no possibility that the assignment to *V2.all* affects the value of *I1*. This means that the

compiler optimizer can infer that the value `I1` is constant for all iterations of the loop and load it from memory only once, before entering the loop, instead of in every iteration (this is called load hoisting).

This kind of optimizations, based on strict type-based aliasing, is triggered by specifying an optimization level of `-O2` or higher (or `-Os`) for the GCC back end and `-O1` or higher for the LLVM back end and allows the compiler to generate more efficient code.

However, although this optimization is always correct in terms of the formal semantics of the Ada Reference Manual, you can run into difficulties arise if you use features like `Unchecked_Conversion` to break the typing system. Consider the following complete program example:

```
package P1 is
  type Int1 is new Integer;
  type A1 is access Int1;

  type Int2 is new Integer;
  type A2 is access Int2;
end P1;

with P1; use P1;
package P2 is
  function To_A2 (Input : A1) return A2;
end p2;

with Ada.Unchecked_Conversion;
package body P2 is
  function To_A2 (Input : A1) return A2 is
    function Conv is
      new Ada.Unchecked_Conversion (A1, A2);
    begin
      return Conv (Input);
    end To_A2;
  end P2;

with P1; use P1;
with P2; use P2;
with Text_IO; use Text_IO;
procedure M is
  V1 : A1 := new Int1;
  V2 : A2 := To_A2 (V1);
begin
  V1.all := 1;
  V2.all := 0;
  Put_Line (Int1'Image (V1.all));
end;
```

This program prints out 0 in `-O0` mode, but it prints out 1 in `-O2` mode. That's because in strict aliasing mode, the compiler may and does assume that the assignment to `V2.all` could not affect the value of `V1.all`, since different types are involved.

This behavior is not a case of non-conformance with the standard, since the Ada RM specifies that an unchecked conversion where the resulting bit pattern is not a correct value of the target type can result in an abnormal value and attempting to reference an abnormal value makes the execution of a program erroneous. That's the case here since the result does not point to an object of type `Int2`. This means that the effect is entirely unpredictable.

However, although that explanation may satisfy a language lawyer, in practice, you probably expect an unchecked conversion involving pointers to create true aliases and the behavior of printing 1 is questionable. In this case, the strict type-based aliasing optimizations are clearly unwelcome.

Indeed, the compiler recognizes this possibility and the instantiation of `Unchecked_Conversion` generates a warning:

```
p2.adb:5:07: warning: possible aliasing problem with type "A2"  
p2.adb:5:07: warning: use -fno-strict-aliasing switch for references  
p2.adb:5:07: warning: or use "pragma No_Strict_Aliasing (A2);"
```

Unfortunately the problem is only recognized when compiling the body of package `P2`, but the actual problematic code is generated while compiling the body of `M` and this latter compilation does not see the suspicious instance of `Unchecked_Conversion`.

As implied by the warning message, there are approaches you can use to avoid the unwanted strict aliasing optimizations in a case like this.

One possibility is to simply avoid the use of higher levels of optimization, but that is quite drastic, since it throws away a number of useful optimizations that don't involve strict aliasing assumptions.

A less drastic approach is for you to compile the program using the `-fno-strict-aliasing` switch. Actually, it is only the unit containing the dereferencing of the suspicious pointer that you need to compile with that switch. So, in this case, if you compile unit `M` with this switch, you get the expected value of 0 printed. Analyzing which units might need the switch can be painful, so you may find it a more reasonable approach is to compile the entire program with options `-O2` and `-fno-strict-aliasing`. If you obtain satisfactory performance with this combination of options, then the advantage is that you have avoided the entire issue of possible problematic optimizations due to strict aliasing.

To avoid the use of compiler switches, you may use the configuration pragma `No_Strict_Aliasing` with no parameters to specify that for all access types, the strict aliasing optimizations should be suppressed.

However, these approaches are still overkill, in that they cause all manipulations of all access values to be deoptimized. A more refined approach is to concentrate attention on the specific access type identified as problematic.

The first possibility is to move the instantiation of unchecked conversion to the unit in which the type is declared. In this example, you would move the instantiation of `Unchecked_Conversion` from the body of package `P2` to the spec of package `P1`. Now, the warning disappears because any use of the access type knows there is a suspicious unchecked conversion and the strict aliasing optimizations are automatically suppressed for it.

If it's not practical to move the unchecked conversion to the same unit in which the destination access type is declared (perhaps because the source type is not visible in that unit),

the second possibility is for you to use `pragma No_Strict_Aliasing` for the type. You must place this pragma in the same declarative part as the declaration of the access type:

```
type A2 is access Int2;
pragma No_Strict_Aliasing (A2);
```

Here again, the compiler now knows that strict aliasing optimizations should be suppressed for any dereference made through type A2 and the expected behavior is obtained.

The third possibility is to declare that one of the designated types involved, namely `Int1` or `Int2`, is allowed to alias any other type in the universe, by using pragma `Universal_Aliasing`:

```
type Int2 is new Integer;
pragma Universal_Aliasing (Int2);
```

The effect is equivalent to applying `pragma No_Strict_Aliasing` to every access type designating `Int2`, in particular `A2`, and, more generally, to every reference made to an object of declared type `Int2`, so it's very powerful and effectively takes `Int2` out of the alias analysis performed by the compiler in all circumstances.

You can also use this pragma used to deal with aliasing issues that arise from the use of `Unchecked_Conversion` in the source code but without the presence of access types. The typical example is code that streams data by means of arrays of storage units (bytes):

```
type Byte is mod 2**System.Storage_Unit;
for Byte'Size use System.Storage_Unit;

type Chunk_Of_Bytes is array (1 .. 64) of Byte;

procedure Send (S : Chunk_Of_Bytes);

type Rec is record
  ...
end record;

procedure Dump (R : Rec) is
  function To_Stream is
    new Ada.Unchecked_Conversion (Rec, Chunk_Of_Bytes);
  begin
    Send (To_Stream (R));
  end;
```

This generates the following warning for the call to `Send`:

```
dump.adb:8:25: warning: unchecked conversion implemented by copy
dump.adb:8:25: warning: use pragma Universal_Aliasing on either type
dump.adb:8:25: warning: to enable RM 13.9(12) implementation permission
```

This occurs because the formal parameter `S` of `Send` is passed by reference by the compiler and it's not possible to pass a reference to `R` directly in the call without violating strict type-based aliasing. That's why the compiler generates a temporary of type `Chunk_Of_Bytes` just before the call and passes a reference to this temporary instead.

As implied by the warning message, you can avoid the temporary (and the warning) by means of pragma `Universal_Aliasing`:

```

type Chunk_Of_Bytes is array (1 .. 64) of Byte;
pragma Universal_Aliasing (Chunk_Of_Bytes);

```

You can also apply this pragma to the component type instead:

```

type Byte is mod 2**System.Storage_Unit;
for Byte'Size use System.Storage_Unit;
pragma Universal_Aliasing (Byte);

```

and every array type whose component is `Byte` will inherit the pragma.

To summarize, the alias analysis performed in strict aliasing mode by the compiler can have significant benefits. We've seen cases of large scale application code where the execution time is increased by up to 5% when these optimizations are turned off. However, if you have code that make significant use of unchecked conversion, you might want to just stick with `-O1` (with the GCC back end) and avoid the entire issue. If you get adequate performance at this level of optimization, that's probably the safest approach. If tests show that you really need higher levels of optimization, then you can experiment with `-O2` and `-O2 -fno-strict-aliasing` to see how much effect this has on size and speed of the code. If you really need to use `-O2` with strict aliasing in effect, then you should review any uses of unchecked conversion, particularly if you are getting the warnings described above.

### 6.3.1.10 Aliased Variables and Optimization

There are scenarios in which your programs may use low level techniques to modify variables that otherwise might be considered to be unassigned. For example, you can pass a variable to a procedure by reference by taking the address of the parameter and using that address to modify the variable's value, even though the address is passed as an `in` parameter. Consider the following example:

```

procedure P is
  Max_Length : constant Natural := 16;
  type Char_Ptr is access all Character;

  procedure Get_String(Buffer: Char_Ptr; Size : Integer);
  pragma Import (C, Get_String, "get_string");

  Name : aliased String (1 .. Max_Length) := (others => ' ');
  Temp : Char_Ptr;

  function Addr (S : String) return Char_Ptr is
    function To_Char_Ptr is
      new Ada.Unchecked_Conversion (System.Address, Char_Ptr);
    begin
      return To_Char_Ptr (S (S'First)'Address);
    end;

begin
  Temp := Addr (Name);
  Get_String (Temp, Max_Length);
end;

```

where `Get_String` is a C function that uses the address in `Temp` to modify the variable `Name`. This code is dubious, and arguably erroneous, and the compiler is entitled to assume that `Name` is never modified, and generate code accordingly.

However, in practice, this could cause some existing code that seems to work with no optimization to start failing at higher levels of optimization.

What the compiler does for such cases, is to assume that marking a variable as aliased indicates that some “funny business” may be going on. The optimizer recognizes the `aliased` keyword and inhibits any optimizations that assume the variable cannot be assigned to. This means that the above example will in fact “work” reliably, that is, it will produce the expected results. However, you should nevertheless avoid code such as this if possible because it’s not portable and may not function as you expect with all compilers.

### 6.3.1.11 Atomic Variables and Optimization

You need to take two things into consideration with regard to performance when you use atomic variables.

First, the RM only guarantees that access to atomic variables be atomic, but has nothing to say about how this is achieved, though there is a strong implication that this should not be achieved by explicit locking code. Indeed, GNAT never generates any locking code for atomic variable access; it will simply reject any attempt to make a variable or type atomic if the atomic access cannot be achieved without such locking code.

That being said, it’s important to understand that you cannot assume the the program will always access the entire variable. Consider this example:

```

type R is record
  A,B,C,D : Character;
end record;
for R'Size use 32;
for R'Alignment use 4;

RV : R;
pragma Atomic (RV);
X : Character;
...
X := RV.B;
```

You cannot assume that the reference to `RV.B` will read the entire 32-bit variable with a single load instruction. It is perfectly legitimate, if the hardware allows it, to do a byte read of just the `B` field. This read is still atomic, which is all the RM requires. GNAT can and does take advantage of this, depending on the architecture and optimization level. Any assumption to the contrary is non-portable and risky. Even if you examine the assembly language and see a full 32-bit load, this might change in a future version of the compiler.

If your application requires that all accesses to `RV` in this example be full 32-bit loads, you need to make a copy for the access as in:

```

declare
  RV_Copy : constant R := RV;
begin
  X := RV_Copy.B;
```

```
end;
```

Now the reference to `RV` must read the whole variable. Actually, one can imagine some compiler which figures out that the whole copy is not required (because only the `B` field is actually accessed), but GNAT certainly won't do that, and we don't know of any compiler that would not handle this right, and the above code will in practice work portably across all architectures (that permit the `Atomic` declaration).

The second issue with atomic variables has to do with the possible requirement of generating synchronization code. For more details on this, consult the sections on the pragmas `Enable/Disable_Atomic_Synchronization` in the :title:GNAT Reference Manual. If performance is critical, and such synchronization code is not required, you may find it useful to disable it.

### 6.3.1.12 Passive Task Optimization

A passive task is one which is sufficiently simple that, in theory, a compiler could recognize it and implement it efficiently without creating a new thread. The original design of Ada 83 had in mind this kind of passive task optimization, but only a few Ada 83 compilers attempted it. The reason was that it was difficult to determine the exact conditions under which the optimization was possible. The result is a very fragile optimization where a very minor change in the program can suddenly silently make a task non-optimizable.

With the revisiting of this issue in Ada 95, there was general agreement that this approach was fundamentally flawed and the notion of protected types was introduced. When using protected types, the restrictions are well defined, you KNOW that the operations will be optimized, and furthermore this optimized performance is fully portable.

Although it would theoretically be possible for GNAT to attempt to do this optimization, it really doesn't make sense in the context of Ada 95 and none of the Ada 95 compilers implement this optimization as far as we know. GNAT never attempts to perform this optimization.

In any new Ada 95 code that you write, you should always use protected types in place of tasks that might be able to be optimized in this manner. Of course, this does not help if you have legacy Ada 83 code that depends on this optimization, but it is unusual to encounter a case where the performance gains from this optimization are significant.

Your program should work correctly without this optimization. If you have performance problems, the most practical approach is to figure out exactly where these performance problems arise and update those particular tasks to be protected types. Note that typically clients of the tasks who call entries will not have to be modified, only the task definitions themselves.

### 6.3.2 Text\_IO Suggestions

The `Ada.Text_IO` package has fairly high overhead due in part to the requirement of maintaining page and line counts. If performance is critical, one recommendation is to use `Stream_IO` instead of `Text_IO` for large-volume output, since it has less overhead.

If you must use `Text_IO`, note that output to the standard output and standard error files is unbuffered by default (this provides better behavior when output statements are used for debugging or if the progress of a program is observed by tracking the output, e.g. by using the Unix `'tail -f'` command to watch redirected output).

If you're generating large volumes of output with `Text_IO` and performance is an important factor, use a designated file instead of the standard output file or change the standard output file to be buffered using `Interfaces.C_Streams.setvbuf`.

### 6.3.3 Reducing Size of Executables with Unused Subprogram/Data Elimination

This section describes how you can eliminate unused subprograms and data from your executable just by setting options at compilation time.

#### 6.3.3.1 About unused subprogram/data elimination

By default, an executable contains all code and data of its objects (directly linked or coming from statically linked libraries), even data or code never used by this executable. This feature eliminates such unused code from your executable, thus making it smaller (in disk and in memory).

You can use this functionality on all Linux platforms except for the IA-64 architecture and on all cross platforms using the ELF binary file format. In both cases, GNU binutils version 2.16 or later are required to enable it.

#### 6.3.3.2 Compilation options

The operation of eliminating the unused code and data from the final executable is directly performed by the linker.

In order to do this, it has to work with objects compiled with the following switches passed to the GCC back end: `-ffunction-sections` `-fdata-sections`.

These options are usable with C and Ada files. They cause the compiler to place each function or data in a separate section in the resulting object file.

Once you've created the objects and static libraries with these switches, the linker can perform the dead code elimination. You can do this by specifying the `-Wl,--gc-sections` switch to your `gcc` command or in the `-largf` section of your invocation of `gnatmake`. This causes the linker to perform a garbage collection and remove code and data that are never referenced.

If the linker performs a partial link (`-r` linker switch), then you need to provide the entry point using the `-e` / `--entry` linker switch.

Note that objects compiled without the `-ffunction-sections` and `-fdata-sections` options can still be linked with the executable. However, no dead code elimination can be performed on those objects (they will be linked as is).

The GNAT static library is compiled with `-ffunction-sections` and `-fdata-sections` on some platforms. This allows you to eliminate the unused code and data of the GNAT library from your executable.

#### 6.3.3.3 Example of unused subprogram/data elimination

Here's a simple example:

```
with Aux;  
  
procedure Test is
```

```

begin
    Aux.Used (10);
end Test;

package Aux is
    Used_Data    : Integer;
    Unused_Data  : Integer;

    procedure Used    (Data : Integer);
    procedure Unused (Data : Integer);
end Aux;

package body Aux is
    procedure Used (Data : Integer) is
    begin
        Used_Data := Data;
    end Used;

    procedure Unused (Data : Integer) is
    begin
        Unused_Data := Data;
    end Unused;
end Aux;

```

`Unused` and `Unused_Data` are never referenced in this code excerpt and hence may be safely removed from the final executable.

```

$ gnatmake test

$ nm test | grep used
020015f0 T aux__unused
02005d88 B aux__unused_data
020015cc T aux__used
02005d84 B aux__used_data

$ gnatmake test -cargs -fdata-sections -ffunction-sections \
    -larges -Wl,--gc-sections

$ nm test | grep used
02005350 T aux__used
0201ffe0 B aux__used_data

```

You can see that the procedure `Unused` and the object `Unused_Data` are removed by the linker when you've used the appropriate switches.

## 6.4 Overflow Check Handling in GNAT

This section explains how to control the handling of overflow checks.

### 6.4.1 Background

Overflow checks are checks that the compiler may make to ensure that intermediate results are not out of range. For example:

```
A : Integer;
...
A := A + 1;
```

If `A` has the value `Integer'Last`, the addition will cause overflow since the result is out of range of the type `Integer`. In this case, execution will raise `Constraint_Error` if checks are enabled.

A trickier situation arises in cases like the following:

```
A, C : Integer;
...
A := (A + 1) + C;
```

where `A` is `Integer'Last` and `C` is `-1`. Here, the final result of the expression on the right hand side is `Integer'Last` which is in range, but the question arises whether the intermediate addition of `(A + 1)` raises an overflow error.

The (perhaps surprising) answer is that the Ada language definition does not answer this question. Instead, it leaves it up to the implementation to do one of two things if overflow checks are enabled.

- \* raise an exception (`Constraint_Error`), or
- \* yield the correct mathematical result which is then used in subsequent operations.

If the compiler chooses the first approach, the execution of this example will indeed raise `Constraint_Error` if overflow checking is enabled or result in erroneous execution if overflow checks are suppressed.

But if the compiler chooses the second approach, it can perform both additions yielding the correct mathematical result, which is in range, so no exception is raised and the right result is obtained, regardless of whether overflow checks are suppressed.

Note that in the first example, an exception will be raised in either case, since if the compiler gives the correct mathematical result for the addition, it will be out of range of the target type of the assignment and thus fails the range check.

This lack of specified behavior in the handling of overflow for intermediate results is a source of non-portability and can thus be problematic when you port programs. Most typically, this arises in a situation where the original compiler did not raise an exception and you move the application to a compiler where the check is performed on the intermediate result and an unexpected exception is raised.

Furthermore, when using Ada 2012's preconditions and other assertion forms, another issue arises. Consider:

```
procedure P (A, B : Integer) with
  Pre => A + B <= Integer'Last;
```

We often want to regard arithmetic in a context such as this from a purely mathematical point of view. So, for example, if the two actual parameters for a call to `P` are both `Integer'Last` then the precondition should be evaluated as `False`. If we're executing in a

mode with run-time checks enabled for preconditions, then we would like this precondition to fail, rather than raising an exception because of the intermediate overflow.

However, the language definition leaves the specification of whether the above condition fails (raising `Assert_Error`) or causes an intermediate overflow (raising `Constraint_Error`) up to the implementation.

The situation is worse in a case such as the following:

```
procedure Q (A, B, C : Integer) with
  Pre => A + B + C <= Integer'Last;
```

Consider the call

```
Q (A => Integer'Last, B => 1, C => -1);
```

From a mathematical point of view the precondition is `True`, but at run time we may (but are not guaranteed to) get an exception raised because of the intermediate overflow (and we really would prefer this precondition to be considered `True` at run time).

## 6.4.2 Management of Overflows in GNAT

To deal with the portability issue and with the problem of mathematical versus run-time interpretation of the expressions in assertions, GNAT provides comprehensive control over the handling of intermediate overflows. It can operate in three modes, and in addition, permits separate selection of operating modes for the expressions within assertions (here the term ‘assertions’ is used in the technical sense, which includes preconditions and so forth) and for expressions appearing outside assertions.

The three modes are:

- \* ‘Use base type for intermediate operations’ (`STRICT`)

In this mode, all intermediate results for predefined arithmetic operators are computed using the base type, and the result must be in range of the base type. If this is not the case, then either an exception is raised (if overflow checks are enabled) or the execution is erroneous (if overflow checks are suppressed). This is the normal default mode.

- \* ‘Most intermediate overflows avoided’ (`MINIMIZED`)

In this mode, the compiler attempts to avoid intermediate overflows by using a larger integer type, typically `Long_Long_Integer`, as the type in which arithmetic is performed for predefined arithmetic operators. This may be slightly more expensive at run time (compared to suppressing intermediate overflow checks), though the cost is negligible on modern 64-bit machines. For the examples given earlier, no intermediate overflows would have resulted in exceptions, since the intermediate results are all in the range of `Long_Long_Integer` (typically 64-bits on nearly all implementations of GNAT). In addition, if checks are enabled, this reduces the number of checks that must be made, so this choice may actually result in an improvement in space and time behavior.

However, there are cases where `Long_Long_Integer` is not large enough. Consider the following example:

```
procedure R (A, B, C, D : Integer) with
  Pre => (A**2 * B**2) / (C**2 * D**2) <= 10;
```

where `A = B = C = D = Integer'Last`. Now the intermediate results are out of the range of `Long_Long_Integer` even though the final result is in range and the precondition is

**True** from a mathematical point of view. In such a case, operating in this mode, an overflow occurs for the intermediate computation (which is why this mode says ‘most’ intermediate overflows are avoided). In this case, an exception is raised if overflow checks are enabled, and the execution is erroneous if overflow checks are suppressed.

\* ‘All intermediate overflows avoided’ (**ELIMINATED**)

In this mode, the compiler avoids all intermediate overflows by using arbitrary precision arithmetic as required. In this mode, the above example with `A**2 * B**2` would not cause intermediate overflow, because the intermediate result would be evaluated using sufficient precision, and the result of evaluating the precondition would be **True**.

This mode has the advantage of avoiding any intermediate overflows, but at the expense of significant run-time overhead, including the use of a library (included automatically in this mode) for multiple-precision arithmetic.

This mode provides cleaner semantics for assertions, since now the run-time behavior emulates true arithmetic behavior for the predefined arithmetic operators, meaning that there is never a conflict between the mathematical view of the assertion and its run-time behavior.

Note that in this mode, the behavior is unaffected by whether or not overflow checks are suppressed, since overflow does not occur. Gigantic intermediate expressions can still raise **Storage\_Error** as a result of attempting to compute the results of such expressions (e.g. `Integer'Last ** Integer'Last`) but overflow is impossible.

Note that these modes apply only to the evaluation of predefined arithmetic, membership, and comparison operators for signed integer arithmetic.

For fixed-point arithmetic, you suppress checks. But if checks are enabled, fixed-point values are always checked for overflow against the base type for intermediate expressions (i.e., such checks always operate in the equivalent of **STRICT** mode).

For floating-point, on nearly all architectures, **Machine\_Overflows** is **False**, and IEEE infinities are generated, so overflow exceptions are never raised. If you want to avoid infinities and check that final results of expressions are in range, you can declare a constrained floating-point type and range checks are carried out in the normal manner (with infinite values always failing all range checks).

### 6.4.3 Specifying the Desired Mode

You can specify the desired mode of for handling intermediate overflow using either the **Overflow\_Mode** pragma or an equivalent compiler switch. The pragma has the form:

```
pragma Overflow_Mode ([General =>] MODE [, [Assertions =>] MODE]);
```

where **MODE** is one of

- \* **STRICT**: intermediate overflows checked (using base type)
- \* **MINIMIZED**: minimize intermediate overflows
- \* **ELIMINATED**: eliminate intermediate overflows

The case is ignored, so **MINIMIZED**, **Minimized** and **minimized** all have the same effect.

If you only specify the **General** parameter, the given **MODE** applies to expressions both within and outside assertions. If you specify both arguments, the value of **General** applies

to expressions outside assertions, and `Assertions` applies to expressions within assertions. For example:

```
pragma Overflow_Mode
  (General => Minimized, Assertions => Eliminated);
```

specifies that expressions outside assertions be evaluated in ‘minimize intermediate overflows’ mode and expressions within assertions be evaluated in ‘eliminate intermediate overflows’ mode. This is often a reasonable choice, avoiding excessive overhead outside assertions, but assuring a high degree of portability when importing code from another compiler while incurring the extra overhead for assertion expressions to ensure that the behavior at run time matches the expected mathematical behavior.

The `Overflow_Mode` pragma has the same scoping and placement rules as pragma `Suppress`, so you can use it either as a configuration pragma, specifying a default for the whole program, or in a declarative scope, where it applies to the remaining declarations and statements in that scope.

Note that pragma `Overflow_Mode` does not affect whether overflow checks are enabled or suppressed. It only controls the method used to compute intermediate values. To control whether overflow checking is enabled or suppressed, use pragma `Suppress` or `Unsuppress` in the usual manner.

Additionally, you can use the compiler switch `-gnato?` or `-gnato??` to control the checking mode default (which you can subsequently override using the above pragmas).

Here `?` is one of the digits 1 through 3:

- |   |   |
|---|---|
| 1 | use base type for intermediate operations ( <code>STRICT</code> ) |
| 2 | minimize intermediate overflows ( <code>MINIMIZED</code> )        |
| 3 | eliminate intermediate overflows ( <code>ELIMINATED</code> )      |

As with the pragma, if only one digit appears, it applies to all cases; if two digits are given, the first applies to expressions outside assertions and the second within assertions. Thus the equivalent of the example pragma above would be `-gnato23`.

If you don’t provide any digits following the `-gnato`, it’s equivalent to `-gnato11`, causing all intermediate operations to be computed using the base type (`STRICT` mode).

#### 6.4.4 Default Settings

The default mode for overflow checks is

```
General => Strict
```

which causes all computations both inside and outside assertions to use the base type, and is equivalent to `-gnato` (with no digits following).

The pragma `Suppress (Overflow_Check)` disables overflow checking but has no effect on the method used for computing intermediate results. The pragma `Unsuppress (Overflow_Check)` enables overflow checking but has no effect on the method used for computing intermediate results.

### 6.4.5 Implementation Notes

In practice, on typical 64-bit machines, the `MINIMIZED` mode is reasonably efficient and you can generally use it. It also helps to ensure compatibility with code imported from other compilers to GNAT.

Setting all intermediate overflows checking (`STRICT` mode) makes sense if you want to make sure your code is compatible with any other Ada implementations. You may find this useful in ensuring portability for code that is to be exported to some other compiler than GNAT.

The Ada standard allows the reassociation of expressions at the same precedence level if no parentheses are present. For example, `A+B+C` parses as though it were `(A+B)+C`, but the compiler can reinterpret this as `A+(B+C)`, possibly introducing or eliminating an overflow exception. The GNAT compiler never takes advantage of this freedom, and the expression `A+B+C` will be evaluated as `(A+B)+C`. If you need the other order, you can write the parentheses explicitly `A+(B+C)` and GNAT will respect this order.

The use of `ELIMINATED` mode will cause the compiler to automatically include an appropriate arbitrary precision integer arithmetic package. The compiler will make calls to this package, though only in cases where it cannot be sure that `Long_Long_Integer` is sufficient to guard against intermediate overflows. This package does not use dynamic allocation, but it does use the secondary stack, so an appropriate secondary stack package must be present (this is always true for standard full Ada, but may require specific steps for restricted run times such as ZFP).

Although `ELIMINATED` mode causes expressions to use arbitrary precision arithmetic, avoiding overflow, the final result must be in an appropriate range. This is true even if the final result is of type `[Long_[Long_]]Integer'Base`, which still has the same bounds as its associated constrained type at run-time.

Currently, the `ELIMINATED` mode is only available on target platforms for which `Long_Long_Integer` is at least 64-bits (nearly all GNAT platforms).

## 6.5 Performing Dimensionality Analysis in GNAT

The GNAT compiler supports dimensionality checking. You can specify physical units for objects and the compiler verifies that uses of these objects are compatible with their dimension, in a fashion that is familiar to engineering practice. The dimensions of algebraic expressions (including powers with static exponents) are computed from their constituents.

This feature depends on Ada 2012 aspect specifications and is available for versions 7.0.1 and later of GNAT. The GNAT-specific aspect `Dimension_System` allows you to define a system of units; the aspect `Dimension` allows you to declare dimensioned quantities within a given system. (These aspects are described in the ‘Implementation Defined Aspects’ chapter of the *GNAT Reference Manual*).

The major advantage of this model is that it does not require the declaration of multiple operators for all possible combinations of types: you only need to use the proper subtypes in object declarations.

The simplest way to impose dimensionality checking on a computation is to make use of one of the instantiations of the package `System.Dim.Generic_Mks`, which is part of the GNAT library. This generic package defines a floating-point type `MKS_Type`, for which a sequence of

dimension names are specified, together with their conventional abbreviations. You should read the following together with the full specification of the package, in file `s-digemk.ads`.

```
type Mks_Type is new Float_Type
with
  Dimension_System => (
    (Unit_Name => Meter,    Unit_Symbol => 'm',    Dim_Symbol => 'L'),
    (Unit_Name => Kilogram, Unit_Symbol => "kg",    Dim_Symbol => 'M'),
    (Unit_Name => Second,   Unit_Symbol => 's',     Dim_Symbol => 'T'),
    (Unit_Name => Ampere,   Unit_Symbol => 'A',     Dim_Symbol => 'I'),
    (Unit_Name => Kelvin,   Unit_Symbol => 'K',     Dim_Symbol => "Theta"),
    (Unit_Name => Mole,     Unit_Symbol => "mol",    Dim_Symbol => 'N'),
    (Unit_Name => Candela,  Unit_Symbol => "cd",    Dim_Symbol => 'J'));
```

The package then defines a series of subtypes that correspond to these conventional units. For example:

```
subtype Length is Mks_Type
with
  Dimension => (Symbol => 'm', Meter => 1, others => 0);
```

and similarly for Mass, Time, Electric\_Current, Thermodynamic\_Temperature, Amount\_Of\_Substance, and Luminous\_Intensity (the standard set of units of the SI system).

The package also defines conventional names for values of each unit, for example:

```
m   : constant Length      := 1.0;
kg  : constant Mass        := 1.0;
s   : constant Time        := 1.0;
A   : constant Electric_Current := 1.0;
```

as well as useful multiples of these units:

```
cm   : constant Length := 1.0E-02;
g    : constant Mass   := 1.0E-03;
min  : constant Time   := 60.0;
day  : constant Time   := 60.0 * 24.0 * min;
...
```

There are three instantiations of `System.Dim.Generic_Mks` defined in the GNAT library:

- \* `System.Dim.Float_Mks` based on `Float` defined in `s-diflmlk.ads`.
- \* `System.Dim.Long_Mks` based on `Long_Float` defined in `s-dilomk.ads`.
- \* `System.Dim.Mks` based on `Long_Long_Float` defined in `s-dimmks.ads`.

Using one of these packages, you can then define a derived unit by providing the aspect that specifies its dimensions within the MKS system as well as the string to be used for output of a value of that unit:

```
subtype Acceleration is Mks_Type
with Dimension => ("m/sec^2",
  Meter => 1,
  Second => -2,
  others => 0);
```

Here's a complete example:

```
with System.Dim.MKS; use System.Dim.Mks;
```

```

with System.Dim.Mks_IO; use System.Dim.Mks_IO;
with Text_IO; use Text_IO;
procedure Free_Fall is
  subtype Acceleration is Mks_Type
    with Dimension => ("m/sec^2", 1, 0, -2, others => 0);
  G : constant acceleration := 9.81 * m / (s ** 2);
  T : Time := 10.0*s;
  Distance : Length;

begin
  Put ("Gravitational constant: ");
  Put (G, Aft => 2, Exp => 0); Put_Line ("");
  Distance := 0.5 * G * T ** 2;
  Put ("distance travelled in 10 seconds of free fall ");
  Put (Distance, Aft => 2, Exp => 0);
  Put_Line ("");
end Free_Fall;

```

Execution of this program yields:

```

Gravitational constant:  9.81 m/sec^2
distance travelled in 10 seconds of free fall 490.50 m

```

However, incorrect assignments such as:

```

Distance := 5.0;
Distance := 5.0 * kg;

```

are rejected with the following diagnoses:

```

Distance := 5.0;
>>> dimensions mismatch in assignment
>>> left-hand side has dimension [L]
>>> right-hand side is dimensionless

Distance := 5.0 * kg;
>>> dimensions mismatch in assignment
>>> left-hand side has dimension [L]
>>> right-hand side has dimension [M]

```

The dimensions of an expression are properly displayed even if there is no explicit subtype for it. If we add to the program:

```

Put ("Final velocity: ");
Put (G * T, Aft =>2, Exp =>0);
Put_Line ("");

```

the output includes:

```

Final velocity: 98.10 m.s**(-1)

```

The type `Mks_Type` is said to be a ‘dimensionable type’ since it has a `Dimension_System` aspect, and the subtypes `Length`, `Mass`, etc., are said to be ‘dimensioned subtypes’ since each one has a `Dimension` aspect.

The `Dimension` aspect of a dimensioned subtype `S` defines a mapping from the base type’s `Unit_Names` to integer (or, more generally, rational) values. This mapping is the ‘dimension

vector' (also referred to as the 'dimensionality') for that subtype, denoted by  $DV(S)$ , and thus for each object of that subtype. Intuitively, the value specified for each `Unit_Name` is the exponent associated with that unit; a zero value means that the unit is not used. For example:

```

declare
  Acc : Acceleration;
  ...
begin
  ...
end;
```

Here  $DV(Acc) = DV(Acceleration) = (Meter=>1, Kilogram=>0, Second=>-2, Ampere=>0, Kelvin=>0, Mole=>0, Candela=>0)$ . Symbolically, we can express this as `Meter / Second**2`.

The dimension vector of an arithmetic expression is synthesized from the dimension vectors of its components, with compile-time dimensionality checks that help prevent mismatches such as using an `Acceleration` where a `Length` is required.

The dimension vector of the result of an arithmetic expression 'expr', or  $DV(expr)$ , is defined as follows, assuming conventional mathematical definitions for the vector operations that are used:

- \* If 'expr' is of the type 'universal\_real', or is not of a dimensioned subtype, then 'expr' is dimensionless;  $DV(expr)$  is the empty vector.
- \*  $DV(op\ expr)$ , where 'op' is a unary operator, is  $DV(expr)$
- \*  $DV(expr1\ op\ expr2)$ , where 'op' is "+" or "-", is  $DV(expr1)$  provided that  $DV(expr1) = DV(expr2)$ . If this condition is not met then the construct is illegal.
- \*  $DV(expr1 * expr2)$  is  $DV(expr1) + DV(expr2)$ , and  $DV(expr1 / expr2) = DV(expr1) - DV(expr2)$ . In this context if one of the 'expr's is dimensionless then its empty dimension vector is treated as (`others => 0`).
- \*  $DV(expr ** power)$  is 'power' \*  $DV(expr)$ , provided that 'power' is a static rational value. If this condition is not met then the construct is illegal.

Note that, by the above rules, it is illegal to use binary "+" or "-" to combine a dimensioned and dimensionless value. Thus an expression such as `acc-10.0` is illegal, where `acc` is an object of subtype `Acceleration`.

The dimensionality checks for relationals use the same rules as for "+" and "-" except when comparing to a literal; thus

```
acc > len
```

is equivalent to

```
acc-len > 0.0
```

and is thus illegal, but

```
acc > 10.0
```

is accepted with a warning. Analogously, a conditional expression requires the same dimension vector for each branch (with no exception for literals).

The dimension vector of a type conversion  $T(\text{expr})$  is defined as follows, based on the nature of  $T$ :

- \* If  $T$  is a dimensioned subtype, then  $DV(T(\text{expr}))$  is  $DV(T)$  provided that either ‘ $\text{expr}$ ’ is dimensionless or  $DV(T) = DV(\text{expr})$ . The conversion is illegal if ‘ $\text{expr}$ ’ is dimensioned and  $DV(\text{expr}) \neq DV(T)$ . Note that vector equality does not require that the corresponding `Unit.Names` be the same.

As a consequence of the above rule, you can convert between different dimension systems that follow the same international system of units, with the seven physical components given in the standard order (length, mass, time, etc.). Thus, you can convert a length in meters to a length in inches (with a suitable conversion factor) but not, for example, to a mass in pounds.

- \* If  $T$  is the base type for ‘ $\text{expr}$ ’ (and the dimensionless root type of the dimension system), then  $DV(T(\text{expr}))$  is  $DV(\text{expr})$ . Thus, if ‘ $\text{expr}$ ’ is of a dimensioned subtype of  $T$ , the conversion may be regarded as a “view conversion” that preserves dimensionality.

This rule means you can write generic code that can be instantiated with compatible dimensioned subtypes. You include in the generic unit conversions that will consequently be present in instantiations, but conversions to the base type will preserve dimensionality and make it possible to write generic code that is correct with respect to dimensionality.

- \* Otherwise (i.e.,  $T$  is neither a dimensioned subtype nor a dimensionable base type),  $DV(T(\text{expr}))$  is the empty vector. Thus, a dimensioned value can be explicitly converted to a non-dimensioned subtype, which of course then escapes dimensionality analysis.

The dimension vector for a type qualification  $T'(\text{expr})$  is the same as for the type conversion  $T(\text{expr})$ .

An assignment statement

```
Source := Target;
```

requires  $DV(\text{Source}) = DV(\text{Target})$  and analogously for parameter passing (the dimension vector for the actual parameter must be equal to the dimension vector for the formal parameter).

When using dimensioned types with elementary functions, you need not instantiate the `Ada.Numerics.Generic_Elementary_Functions` package using the `Mks_Type` nor for any of the derived subtypes such as `Distance`. For functions such as `Sqrt`, the dimensional analysis will fail when using the subtypes because both the parameter and return are of the same type.

An example instantiation

```
package Mks_Numerics is new
  Ada.Numerics.Generic_Elementary_Functions (System.Dim.Mks.Mks_Type);
```

## 6.6 Stack Related Facilities

This section describes some useful tools associated with stack checking and analysis. In particular, it deals with dynamic and static stack usage measurements.

### 6.6.1 Stack Overflow Checking

For most operating systems, `gcc` does not perform stack overflow checking by default. This means that if the main environment task or some other task exceeds the available stack space, unpredictable behavior will occur. Most native systems offer some level of protection by adding a guard page at the end of each task stack. This mechanism is usually not enough for dealing properly with stack overflow situations because a large local variable could “jump” above the guard page. Furthermore, when the guard page is hit, there may not be any space left on the stack for executing the exception propagation code. Enabling stack checking avoids such situations.

To activate stack checking, compile all units with the `gcc` switch `-fstack-check`. For example:

```
$ gcc -c -fstack-check package1.adb
```

Units compiled with this option will generate extra instructions to check that any use of the stack (for procedure calls or for declaring local variables in declare blocks) does not exceed the available stack space. If the space is exceeded, a `Storage_Error` exception is raised.

For declared tasks, the default stack size is defined by the GNAT runtime, whose size may be modified at bind time through the `-d` bind switch ([Switches for gnatbind], page 156). You can set task specific stack sizes using the `Storage_Size` pragma.

For the environment task, the stack size is determined by the operating system. Consequently, to modify the size of the environment task please refer to your operating system documentation.

When using the LLVM back end, this switch doesn’t perform full stack overflow checking, but just checks for very large local dynamic allocations.

### 6.6.2 Static Stack Usage Analysis

A unit compiled with the `-fstack-usage` switch generate an extra file that specifies the maximum amount of stack used on a per-function basis. The file has the same basename as the target object file with a `.su` extension. Each line of this file is made up of three fields:

- \* The name of the function.
- \* A number of bytes.
- \* One or more qualifiers: `static`, `dynamic`, `bounded`.

The second field corresponds to the size of the known part of the function frame.

The qualifier `static` means that the function frame size is purely static. It usually means that all local variables have a static size. In this case, the second field is a reliable measure of the function stack utilization.

The qualifier `dynamic` means that the function frame size is not static. It happens mainly when some local variables have a dynamic size. When this qualifier appears alone, the second field is not a reliable measure of the function stack analysis. When it is qualified with `bounded`, it means that the second field is a reliable maximum of the function stack utilization.

Compilation of a unit with the `-Wstack-usage` switch will issue a warning for each subprogram whose stack usage might be larger than the specified amount of bytes. The wording of that warning is consistent with that in the file documented above.

This is not supported by the LLVM back end.

### 6.6.3 Dynamic Stack Usage Analysis

You can measure the maximum amount of stack used by a task by adding a switch to `gnatbind`, as:

```
$ gnatbind -u0 file
```

With this option, at each task termination, its stack usage is output on `stderr`. Note that this switch is not compatible with tools like Valgrind and DrMemory; they will report errors.

It is not always convenient to output the stack usage when the program is still running. Hence, you can delay this output until the termination of the number of tasks specified as the argument of the `-u` switch. For example:

```
$ gnatbind -u100 file
```

buffers the stack usage information of the first 100 tasks to terminate and outputs it when the program terminates. Results are displayed in four columns:

```
Index | Task Name | Stack Size | Stack Usage
```

where:

- \* ‘Index’ is a number associated with each task.
- \* ‘Task Name’ is the name of the task analyzed.
- \* ‘Stack Size’ is the maximum size for the stack.
- \* ‘Stack Usage’ is the measure done by the stack analyzer. In order to prevent overflow, the stack is not entirely analyzed, and it’s not possible to know exactly how much has actually been used.

By default, `gnatbind` does not process the environment task stack, the stack that contains the main unit. To enable processing of the environment task stack, set the environment variable `GNAT_STACK_LIMIT` to the maximum size of the environment task stack. This amount is given in kilobytes. For example:

```
$ set GNAT_STACK_LIMIT 1600
```

would specify to the analyzer that the environment task stack has a limit of 1.6 megabytes. Any stack usage beyond this will be ignored by the analysis.

This is not supported by the LLVM back end.

The package `GNAT.Task_Stack_Usage` provides facilities to get stack-usage reports at run time. See its body for the details.

## 6.7 Memory Management Issues

This section describes some useful memory pools provided in the GNAT library, and in particular the GNAT Debug Pool facility, which can be used to detect incorrect uses of access values (including ‘dangling references’).

### 6.7.1 Some Useful Memory Pools

The `System.Pool_Global` package provides the `Unbounded_No_Reclaim_Pool` storage pool. Allocations use the standard system call `malloc` while deallocations use the standard system call `free`. No reclamation is performed when the pool goes out of scope. For performance reasons, the standard default Ada allocators/deallocators do not use any explicit storage

pools but if they did, they could use this storage pool without any change in behavior. That is why this storage pool is used when the user makes the default implicit allocator explicit as in this example:

```

type T1 is access Something;
-- no Storage pool is defined for T2

type T2 is access Something_Else;
for T2'Storage_Pool use T1'Storage_Pool;
-- the above is equivalent to
for T2'Storage_Pool use System.Pool_Global.Global_Pool_Object;
```

The `System.Pool_Local` package provides the `Unbounded_Reclaim_Pool` storage pool. Its allocation strategy is similar to `Pool_Local` except that the all storage allocated with this pool is reclaimed when the pool object goes out of scope. This pool provides a explicit mechanism similar to the implicit one provided by several Ada 83 compilers for allocations performed through a local access type and whose purpose was to reclaim memory when exiting the scope of a given local access. As an example, the following program does not leak memory even though it does not perform explicit deallocation:

```

with System.Pool_Local;
procedure Pooloc1 is
  procedure Internal is
    type A is access Integer;
    X : System.Pool_Local.Unbounded_Reclaim_Pool;
    for A'Storage_Pool use X;
    v : A;
  begin
    for I in 1 .. 50 loop
      v := new Integer;
    end loop;
  end Internal;
begin
  for I in 1 .. 100 loop
    Internal;
  end loop;
end Pooloc1;
```

The `System.Pool_Size` package implements the `Stack_Bounded_Pool` used when `Storage_Size` is specified for an access type. The whole storage for the pool is allocated at once, usually on the stack at the point where the access type is elaborated. It is automatically reclaimed when exiting the scope where the access type is defined. This package is not intended to be used directly by the user; it is implicitly used for each declaration with a specified `Storage_Size`:

```

type T1 is access Something;
for T1'Storage_Size use 10_000;
```

### 6.7.2 The GNAT Debug Pool Facility

Using unchecked deallocation and unchecked conversion can easily lead to incorrect memory references. The problems generated by such references are usually difficult to find because

the symptoms can be very remote from the origin of the problem. In such cases, it is very helpful to detect the problem as early as possible. This is the purpose of the Storage Pool provided by `GNAT.Debug_Pools`.

In order to use the GNAT specific debugging pool, you must associate a debug pool object with each of the access types that may be related to suspected memory problems. See Ada Reference Manual 13.11.

```
type Ptr is access Some_Type;
Pool : GNAT.Debug_Pools.Debug_Pool;
for Ptr'Storage_Pool use Pool;
```

`GNAT.Debug_Pools` is derived from a GNAT-specific kind of pool: the `Checked_Pool`. Such pools, like standard Ada storage pools, allow you to redefine allocation and deallocation strategies. They also provide a checkpoint for each dereference through the use of the primitive operation `Dereference` which is implicitly called at each dereference of an access value.

Once you have associated an access type with a debug pool, operations on values of the type may raise four distinct exceptions, which correspond to four potential kinds of memory corruption:

- \* `GNAT.Debug_Pools.Accessing_Not_Allocated_Storage`
- \* `GNAT.Debug_Pools.Accessing_Deallocated_Storage`
- \* `GNAT.Debug_Pools.Freeing_Not_Allocated_Storage`
- \* `GNAT.Debug_Pools.Freeing_Deallocated_Storage`

For types associated with a `Debug_Pool`, dynamic allocation is performed using the standard GNAT allocation routine. References to all allocated chunks of memory are kept in an internal dictionary. Several deallocation strategies are provided, allowing you to choose to release the memory to the system, keep it allocated for further invalid access checks, or fill it with an easily recognizable pattern for debug sessions. The memory pattern is the old IBM hexadecimal convention: `16#DEADBEEF#`.

See the documentation in the file `g-debpoo.ads` for more information on the various strategies.

Upon each dereference, a check is made that the access value denotes a properly allocated memory location. Here's a complete example of use of `Debug_Pools`, which includes typical instances of memory corruption:

```
with GNAT.IO; use GNAT.IO;
with Ada.Unchecked_Deallocation;
with Ada.Unchecked_Conversion;
with GNAT.Debug_Pools;
with System.Storage_Elements;
with Ada.Exceptions; use Ada.Exceptions;
procedure Debug_Pool_Test is

    type T is access Integer;
    type U is access all T;

    P : GNAT.Debug_Pools.Debug_Pool;
```

```

    for T'Storage_Pool use P;

    procedure Free is new Ada.Unchecked_Deallocation (Integer, T);
    function UC is new Ada.Unchecked_Conversion (U, T);
    A, B : aliased T;

    procedure Info is new GNAT.Debug_Pools.Print_Info(Put_Line);

begin
    Info (P);
    A := new Integer;
    B := new Integer;
    B := A;
    Info (P);
    Free (A);
    begin
        Put_Line (Integer'Image(B.all));
    exception
        when E : others => Put_Line ("raised: " & Exception_Name (E));
    end;
    begin
        Free (B);
    exception
        when E : others => Put_Line ("raised: " & Exception_Name (E));
    end;
    B := UC(A'Access);
    begin
        Put_Line (Integer'Image(B.all));
    exception
        when E : others => Put_Line ("raised: " & Exception_Name (E));
    end;
    begin
        Free (B);
    exception
        when E : others => Put_Line ("raised: " & Exception_Name (E));
    end;
    Info (P);
end Debug_Pool_Test;

```

The debug pool mechanism provides the following precise diagnostics on the execution of this erroneous program:

```

Debug Pool info:
Total allocated bytes : 0
Total deallocated bytes : 0
Current Water Mark: 0
High Water Mark: 0

```

```

Debug Pool info:
  Total allocated bytes : 8
  Total deallocated bytes : 0
  Current Water Mark: 8
  High Water Mark: 8

raised: GNAT.DEBUG_POOLS.ACCESSING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.ACCESSING_NOT_ALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_NOT_ALLOCATED_STORAGE
Debug Pool info:
  Total allocated bytes : 8
  Total deallocated bytes : 4
  Current Water Mark: 4
  High Water Mark: 8

```

## 6.8 Sanitizers for Ada

This section explains how to use sanitizers with Ada code. Sanitizers offer code instrumentation and run-time libraries that detect certain memory issues and undefined behaviors during execution. They provide dynamic analysis capabilities useful for debugging and testing.

While many sanitizer capabilities overlap with Ada's built-in runtime checks, they are particularly valuable for identifying issues that arise from unchecked features or low-level operations.

### 6.8.1 AddressSanitizer

AddressSanitizer (aka ASan) is a memory error detector activated with the `-fsanitize=address` switch. Note that many of the typical memory errors, such as use after free or buffer overflow, are detected by Ada's `Access_Check` and `Index_Check`.

It can detect the following types of problems:

- \* Wrong memory overlay

A memory overlay is a situation in which an object of one type is placed at the same memory location as a distinct object of a different type, thus overlaying one object over the other in memory. When there is an overflow because the objects do not overlap (like in the following example), the sanitizer can signal it.

```

procedure Wrong_Size_Overlay is
  type Block is array (Natural range <>) of Integer;

  Block4 : aliased Block := (1 .. 4 => 4);
  Block5 : Block (1 .. 5) with Address => Block4'Address;
begin
  Block5 (Block5'Last) := 5; -- Outside the object
end Wrong_Size_Overlay;

```

If the code is built with the `-fsanitize=address` and `-g` options, the following error is shown at execution time:

```
...
SUMMARY: AddressSanitizer: stack-buffer-overflow wrong_size_overlay.adb:
...
```

#### \* Buffer overflow

Ada's `Index_Check` detects buffer overflows caused by out-of-bounds array access. If run-time checks are disabled, the sanitizer can still detect such overflows at execution time the same way as it signalled the previous wrong memory overlay. Note that if both the Ada run-time checks and the sanitizer are enabled, the Ada run-time exception takes precedence.

```
procedure Buffer_Overrun is
  Size : constant := 100;
  Buffer : array (1 .. Size) of Integer := (others => 0);
  Wrong_Index : Integer := Size + 1 with Export;
begin
  -- Access outside the boundaries
  Put_Line ("Value: " & Integer'Image (Buffer (Wrong_Index)));
end Buffer_Overrun;
```

#### \* Use after lifetime

Ada's `Accessibility_Check` helps prevent use-after-return and use-after-scope errors by enforcing lifetime rules. When these checks are bypassed using `Unchecked_Access`, sanitizers can still detect such violations during execution.

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

procedure Use_After_Return is
  type Integer_Access is access all Integer;
  Ptr : Integer_Access;

  procedure Inner;

  procedure Inner is
    Local : aliased Integer := 42;
  begin
    Ptr := Local'Unchecked_Access;
  end Inner;

begin
  Inner;
  -- Accessing Local after it has gone out of scope
  Put_Line ("Value: " & Integer'Image (Ptr.all));
end Use_After_Return;
```

If the code is built with the `-fsanitize=address` and `-g` options, the following error is shown at execution time:

```
...
==1793927==ERROR: AddressSanitizer: stack-use-after-return on address 0x
READ of size 4 at 0xf6fa1a409060 thread T0
```

```

#0 0xb20b6cb6cabc in _ada_use_after_return use_after_return.adb:18
...

Address 0xf6fa1a409060 is located in stack of thread T0 at offset 32 in
#0 0xb20b6cb6c794 in use_after_return__inner use_after_return.adb:9

This frame has 1 object(s):
[32, 36) 'local' (line 10) <== Memory access at offset 32 is inside
SUMMARY: AddressSanitizer: stack-use-after-return use_after_return.adb:1
...

```

#### \* Memory leak

A memory leak happens when a program allocates memory from the heap but fails to release it after it is no longer needed and loses all references to it like in the following example.

```

procedure Memory_Leak is
  type Integer_Access is access Integer;

  procedure Allocate is
    Ptr : Integer_Access := new Integer'(42);
  begin
    null;
  end Allocate;
begin
  -- Memory leak occurs in the following procedure
  Allocate;
end Memory_Leak;

```

If the code is built with the `-fsanitize=address` and `-g` options, the following error is emitted at execution time showing the location of the offending allocation.

```

==1810634==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
#0 0xe3cbee4bb4a8 in __interceptor_malloc asan_malloc_linux.cpp:69
#1 0xc15bb25d0af8 in __gnat_malloc (memory_leak+0x10af8) (BuildId: f
#2 0xc15bb25c9060 in memory_leak__allocate memory_leak.adb:5
...

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).

```

### 6.8.2 UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (aka UBSan) modifies the program at compile-time to catch various kinds of undefined behavior during program execution.

Different sanitize options (`-fsanitize=alignment,float-cast-overflow,signed-integer-overflow`) detect the following types of problems:

#### \* Wrong alignment

The `-fsanitize=alignment` flag (included also in `-fsanitize=undefined`) enables run-time checks for misaligned memory accesses, ensuring that objects are accessed at addresses that conform to the alignment constraints of their declared types. Violations may lead to crashes or performance penalties on certain architectures.

In the following example:

```
with Ada.Text_IO; use Ada.Text_IO;
with System.Storage_Elements; use System.Storage_Elements;

procedure Misaligned_Address is
  type Aligned_Integer is new Integer with
    Alignment => 4; -- Ensure 4-byte alignment

  Reference : Aligned_Integer := 42; -- Properly aligned object

  -- Create a misaligned object by modifying the address manually
  Misaligned : Aligned_Integer with Address => Reference'Address + 1;

begin
  -- This causes undefined behavior or an alignment exception on strict
  Put_Line ("Misaligned Value: " & Aligned_Integer'Image (Misaligned));
end Misaligned_Address;
```

If the code is built with the `-fsanitize=alignment` and `-g` options, the following error is shown at execution time.

```
misaligned_address.adb:15:51: runtime error: load of misaligned address
```

#### \* Signed integer overflow

Ada performs range checks at runtime in arithmetic operation on signed integers to ensure the value is within the target type's bounds. If this check is removed, the `-fsanitize=signed-integer-overflow` flag (included also in `-fsanitize=undefined`) enables run-time checks for signed integer overflows.

In the following example:

```
procedure Signed_Integer_Overflow is
  type Small_Int is range -128 .. 127;
  X, Y, Z : Small_Int with Export;
begin
  X := 100;
  Y := 50;
  -- This addition will exceed 127, causing an overflow
  Z := X + Y;
end Signed_Integer_Overflow;
```

If the code is built with the `-fsanitize=signed-integer-overflow` and `-g` options, the following error is shown at execution time.

```
signed_integer_overflow.adb:8:11: runtime error: signed integer overflow
```

#### \* Float to integer overflow

When converting a floating-point value to an integer type, Ada performs a range check at runtime to ensure the value is within the target type's bounds. If this check is

removed, the sanitizer can detect overflows in conversions from floating point to integer types.

In the following code:

```

procedure Float_Cast_Overflow is
  Flt : Float := Float'Last with Export;
  Int : Integer;
begin
  Int := Integer (Flt); -- Overflow
end Float_Cast_Overflow;

```

If the code is built with the `-fsanitize=float-cast-overflow` and `-g` options, the following error is shown at execution time.

```
float_cast_overflow.adb:5:20: runtime error: 3.40282e+38 is outside the
```

### 6.8.3 Sanitizers in mixed-language applications

Most of the checks performed by sanitizers operate at a global level, which means they can detect issues even when they span across language boundaries. This applies notably to:

- \* All checks performed by the AddressSanitizer: wrong memory overlays, buffer overflows, uses after lifetime, memory leaks. These checks apply globally, regardless of where the objects are allocated or defined, or where they are destroyed
- \* Wrong alignment checks performed by the UndefinedBehaviorSanitizer. It will check whether an object created in a given language is accessed in another with an incompatible alignment

An interesting case that highlights the benefit of global sanitization is a buffer overflow caused by a mismatch in language bindings. Consider the following C function, which allocates an array of 4 characters:

```

char *get_str (void) {
  char *str = malloc (4 * sizeof (char));
}

```

This function is then bound to Ada code, which incorrectly assumes the buffer is of size 5:

```

type Buffer is array (1 .. 5) of Character;

function Get_Str return access Buffer
  with Import => True, Convention => C, External_Name => "get_str";

Str : access Buffer := Get_Str;
Ch  : Character    := S (S'Last); -- Detected by AddressSanitizer as error

```

On the Ada side, accessing `Str (5)` appears valid because the array type declares five elements. However, the actual memory allocated in C only holds four. This mismatch is not detectable by Ada run-time checks, because Ada has no visibility into how the memory was allocated.

However, the AddressSanitizer will detect the heap buffer overflow at runtime, halting execution and providing a clear diagnostic:

```

...
SUMMARY: AddressSanitizer: heap-buffer-overflow buffer_overflow.adb:20 in _a

```

...

## 7 Platform-Specific Information

This appendix contains information relating to the implementation of run-time libraries on various platforms. It also covers topics related to the GNAT implementation on specific Operating Systems.

### 7.1 Run-Time Libraries

The GNAT run-time implementation may vary with respect to both the underlying threads library and the exception-handling scheme. For threads support, the default run-time will bind to the thread package of the underlying operating system.

For exception handling, either or both of two models are supplied:

- \* ‘Zero-Cost Exceptions’ (“ZCX”), which uses binder-generated tables that are interrogated at run time to locate a handler.
- \* ‘setjmp / longjmp’ (‘SJLJ’), which uses dynamically-set data to establish the set of handlers

Most programs should experience a substantial speed improvement by being compiled with a ZCX run-time. This is especially true for tasking applications or applications with many exception handlers. Note however that the ZCX run-time does not support asynchronous abort of tasks (`abort` and `select-then-abort` constructs) and will instead implement abort by polling points in the runtime. You can also add additional polling points explicitly if needed in your application via `pragma Abort_Defer`.

This section summarizes which combinations of threads and exception support are supplied on various GNAT platforms.

#### 7.1.1 Summary of Run-Time Configurations

Platform	Run-Time	Tasking	Exceptions
GNU/Linux	rts-native (default)	pthread library	ZCX
rts-sjlj	pthread library	SJLJ	
Windows	rts-native (default)	native Win32 threads	ZCX
rts-sjlj	native threads	Win32 SJLJ	
Mac OS	rts-native	pthread library	ZCX

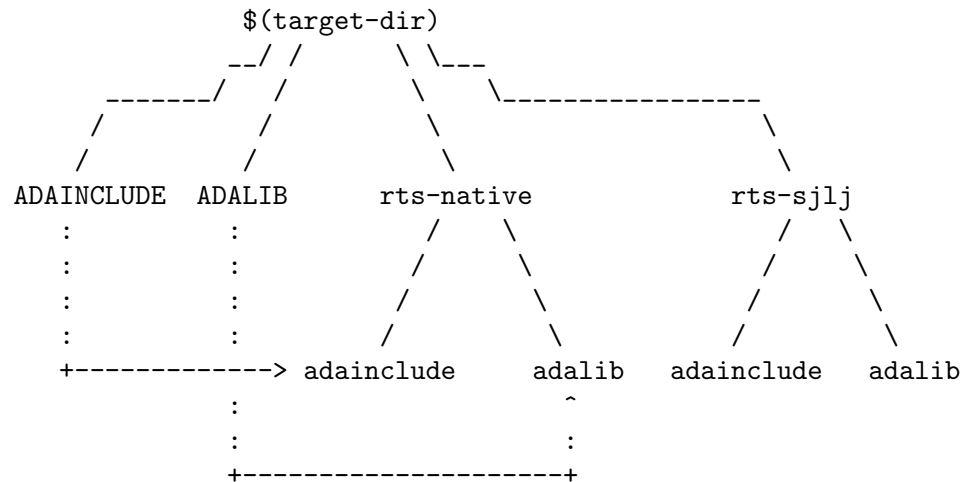
### 7.2 Specifying a Run-Time Library

The `adainclude` subdirectory containing the sources of the GNAT run-time library and the `adalib` subdirectory containing the ALI files and the static and/or shared GNAT library are located in the gcc target-dependent area:

```
target=$prefix/lib/gcc/gcc-*dumpmachine*/gcc-*dumpversion*/
```

As indicated above, on some platforms, several run-time libraries are supplied. These libraries are installed in the target dependent area and contain a complete source and binary subdirectory. The detailed description below explains the differences between the different libraries in terms of their thread support.

The default run-time library (when GNAT is installed) is ‘rts-native’. This default run-time is selected by the means of soft links. For example on x86-linux:



#### Run-Time Library Directory Structure

(Upper-case names and dotted/dashed arrows represent soft links)

If you want to select the ‘rts-sjlj’ library on a permanent basis, you can modify these soft links with the following commands:

```

$ cd $target
$ rm -f adainclude adalib
$ ln -s rts-sjlj/adainclude adainclude
$ ln -s rts-sjlj/adalib adalib

```

Alternatively, you can specify `rts-sjlj/adainclude` in the file `$target/ada_source_path` and `rts-sjlj/adalib` in `$target/ada_object_path`.

You can select another run-time library temporarily by using the `--RTS` switch, e.g., `--RTS=sjlj`

## 7.3 GNU/Linux Topics

This section describes topics that are specific to GNU/Linux platforms.

### 7.3.1 Required Packages on GNU/Linux

GNAT requires the C library developer’s package to be installed. The name of that package depends on your GNU/Linux distribution:

- \* RedHat, SUSE: `glibc-devel`;
- \* Debian, Ubuntu: `libc6-dev` (normally installed by default).

If you’re using the 32-bit version of GNAT on a 64-bit version of GNU/Linux, you’ll need the 32-bit version of the following packages:

- \* RedHat, SUSE: `glibc.i686`, `glibc-devel.i686`, `ncurses-libs.i686`
- \* SUSE: `glibc-locale-base-32bit`
- \* Debian, Ubuntu: `libc6:i386`, `libc6-dev:i386`, `lib32ncursesw5`

Other GNU/Linux distributions might choose different name for those packages.

### 7.3.2 Position Independent Executable (PIE) Enabled by Default on Linux

GNAT generates Position Independent Executable (PIE) code by default. PIE binaries are loaded into random memory locations, introducing an additional layer of protection against attacks.

Building PIE binaries requires that all of their dependencies also be built as Position Independent. If the link of your project fails with an error like:

```
/[...]/ld: /path/to/object/file: relocation R_X86_64_32S against symbol
`symbol name' can not be used when making a PIE object;
recompile with -fPIE
```

it means the identified object file has not been built as Position Independent.

If you are not interested in building PIE binaries, you can simply turn this feature off by first compiling your code with `-fno-pie` and then by linking with `-no-pie` (note the subtle but important difference in the names of the switches – the linker switch does ‘not’ have an *f* after the dash!). When using `gprbuild`, you do this by updating the ‘Required\_Switches’ attribute in package *Compiler* and, depending on your type of project, either attribute ‘Switches’ or attribute ‘Library\_Options’ in package *Linker*.

On the other hand, if you would like to build PIE binaries and you are getting the error above, a quick and easy workaround to allow linking to succeed again is to disable PIE during the link, thus temporarily lifting the requirement that all dependencies also be Position Independent code. To do so, you simply need to add `-no-pie` to the list of switches passed to the linker. As part of this workaround, there is no need to adjust the compiler switches.

From there, to be able to link your binaries with PIE and therefore drop the `-no-pie` workaround, you’ll need to get the identified dependencies rebuilt with PIE enabled (compiled with `-fPIE` and linked with `-pie`).

### 7.3.3 Choosing the Scheduling Policy with GNU/Linux

When using a POSIX threads implementation, you have a choice of several scheduling policies: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`.

Typically, the default is `SCHED_OTHER`, while using `SCHED_FIFO` or `SCHED_RR` requires special (e.g., root) privileges.

By default, GNAT uses the `SCHED_OTHER` policy. To specify `SCHED_FIFO`, you can use one of the following:

- \* `pragma Time_Slice (0.0)`
- \* the corresponding binder switch `-T0`
- \* `pragma Task_Dispatching_Policy (FIFO_Within_Priorities)`

To specify `SCHED_RR`, you should use `pragma Time_Slice` with a value greater than 0.0, or else use the corresponding `-T` binder switch.

To make sure a program is running as root, you can put something like this in a library package body in your application:

```
function geteuid return Integer;
pragma Import (C, geteuid, "geteuid");
Ignore : constant Boolean :=
    (if geteuid = 0 then True else raise Program_Error with "must be root");
```

This gets the effective user id and if it's not 0 (i.e. root), it raises `Program_Error`. Note that if you're running the code in a container, this may not be sufficient as you may have sufficient privilege on the container, but not on the host machine running the container, so check that you also have sufficient privilege for running the container image.

### 7.3.4 A GNU/Linux Debug Quirk

On SuSE 15, some kernels have a defect causing issues when debugging programs using threads or Ada tasks. Due to the lack of documentation found regarding this kernel issue, we can only provide limited information about which kernels are impacted. Kernel version 5.3.18 is known to be impacted and kernels in the 5.14 range or newer are believed to fix this problem.

The bug affects the debugging of 32-bit processes on a 64-bit system. Symptoms can vary: Unexpected `SIGABRT` signals being received by the program, “The futex facility returned an unexpected error code” error message, and inferior programs hanging indefinitely range among the symptoms most commonly observed.

## 7.4 Microsoft Windows Topics

This section describes topics that are specific to the Microsoft Windows platforms.

### 7.4.1 Using GNAT on Windows

One of the strengths of the GNAT technology is that its tool set (`gcc`, `gnatbind`, `gnatlink`, `gnatmake`, the `gdb` debugger, etc.) is used in the same way regardless of the platform.

On Windows, this tool set is complemented by a number of Microsoft-specific tools that have been provided to facilitate interoperability with Windows when this is required. With these tools:

- \* You can build applications using the `CONSOLE` or `WINDOWS` subsystems.
- \* You can use any Dynamically Linked Library (DLL) in your Ada code (both relocatable and non-relocatable DLLs are supported).
- \* You can build Ada DLLs for use in other applications. You can write these applications in a language other than Ada (e.g., C, C++, etc). Again, both relocatable and non-relocatable Ada DLLs are supported.
- \* You can include Windows resources in your Ada application.
- \* You can use or create COM/DCOM objects.

Listed immediately below are all known general GNAT-for-Windows restrictions. We list other restrictions about specific features such as Windows Resources and DLLs in separate sections below.

- \* You cannot use `GetLastError` and `SetLastError` when tasking, protected records, or exceptions are used. In these cases, in order to implement Ada semantics, the GNAT

run-time system calls certain Win32 routines that set the last error variable to 0 upon success. You may be able to use `GetLastError` and `SetLastError` when tasking, protected record, and exception features are not used, but it is not guaranteed to work.

- \* You cannot link against Microsoft C++ libraries except for import libraries. You must do interfacing by means of DLLs.
- \* You can link against Microsoft C libraries. However, the preferred solution is to use C/C++ compiler that comes with GNAT, since it doesn't require having two different development environments and makes the inter-language debugging experience smoother.
- \* When the compilation environment is located on FAT32 drives, you may experience recompilations of source files that have not changed if Daylight Saving Time (DST) state has changed since the last time files were compiled. NTFS drives do not have this problem.
- \* No components of the GNAT toolset use any entries in the Windows registry. The only entries installation of GNAT may create are file associations and PATH settings, provided you chose to create them at installation time, as well as some minimal book-keeping information needed to correctly uninstall or integrate different GNAT products.

### 7.4.2 Using a network installation of GNAT

Make sure the system on which GNAT is installed is accessible from the current machine, i.e., the install location is shared over the network. Shared resources are accessed on Windows by means of UNC paths, which have the format `\\\\server\\sharename\\path`

In order to use such a network installation, simply add the UNC path of the `bin` directory of your GNAT installation in front of your PATH. For example, if GNAT is installed in `\\GNAT` directory of a share location called `c-drive` on a machine `LOKI`, the following command will make it available:

```
$ path \\\loki\c-drive\gnat\bin;%path%
```

Be aware that every compilation using the network installation results in the transfer of large amounts of data across the network and will likely cause a serious performance penalty.

### 7.4.3 CONSOLE and WINDOWS subsystems

There are two main subsystems under Windows. The `CONSOLE` subsystem (which is the default subsystem) always creates a console when launching the application. This is not something desirable when the application has a Windows GUI. To remove this console, your application must use the `WINDOWS` subsystem. To do so, you must specify the `-mwindows` linker switch.

```
$ gnatmake winprog -larges -mwindows
```

### 7.4.4 Temporary Files

You can control where temporary files get created by setting the `TMP` environment variable. The file will be created:

- \* Under the directory pointed to by the `TMP` environment variable if this directory exists.
- \* Under `c:\temp`, if the `TMP` environment variable is not set (or not pointing to a directory) and if this directory exists.
- \* Under the current working directory otherwise.

This allows you to determine exactly where the temporary file will be created. This is particularly useful in networked environments where you may not have write access to some directories.

### 7.4.5 Disabling Command Line Argument Expansion

By default, an executable compiled for the Windows platform will do the following post-processing on the arguments passed on the command line:

- \* If the argument contains the characters `*` and/or `?`, file expansion will be attempted. For example, if the current directory contains `a.txt` and `b.txt`, then when calling:

```
$ my_ada_program *.txt
```

The following arguments will effectively be passed to the main program (for example when using `Ada.Command_Line.Argument`):

```
Ada.Command_Line.Argument (1) -> "a.txt"
```

```
Ada.Command_Line.Argument (2) -> "b.txt"
```

- \* You can disable filename expansion for a given argument by using single quotes. Thus, calling:

```
$ my_ada_program '*.txt'
```

will result in:

```
Ada.Command_Line.Argument (1) -> "*.txt"
```

Note that if the program is launched from a shell such as Cygwin Bash, quote removal might be performed by that shell.

In some contexts, it might be useful to disable this feature (for example if the program performs its own argument expansion). In order to do this, a C symbol needs to be defined and set to 0. You can do this by adding the following code fragment in one of your Ada units:

```
Do_Argv_Expansion : Integer := 0;
pragma Export (C, Do_Argv_Expansion, "__gnat_do_argv_expansion");
```

The results of previous examples will be respectively:

```
Ada.Command_Line.Argument (1) -> "*.txt"
```

and:

```
Ada.Command_Line.Argument (1) -> "'.txt'"
```

### 7.4.6 Choosing the Scheduling Policy with Windows

Under Windows, the standard 31 priorities of the Ada model are mapped onto Window's seven standard priority levels by default: Idle, Lowest, Below Normal, Normal, Above Normal,

When using the `FIFO_Within_Priorities` task dispatching policy, GNAT assigns the `REALTIME_PRIORITY_CLASS` priority class to the application and maps the Ada priority range to the sixteen priorities made available under `REALTIME_PRIORITY_CLASS`.

For details on the values of the different priority mappings, see declarations in `system.ads`. For more information about Windows priorities, please refer to Microsoft documentation.

### 7.4.7 Windows Socket Timeouts

Microsoft Windows desktops older than 8.0 and Microsoft Windows Servers older than 2019 set a socket timeout 500 milliseconds longer than the value set by `setsockopt` with `SO_RCVTIMEO` and `SO_SNDTIMEO` options. The GNAT runtime makes a correction for the difference in the corresponding Windows versions. For Windows Server starting with version 2019, you must provide a manifest file for the GNAT runtime to be able to recognize that the Windows version does not need the timeout correction. The manifest file should be located in the same directory as the executable file and its file name must match the executable name suffixed by `.manifest`. For example, if the executable name is `sock_wto.exe`, the manifest file name must be `sock_wto.exe.manifest`. The manifest file must contain at least the following data:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
<application>
  <!-- Windows Vista -->
  <supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>
  <!-- Windows 7 -->
  <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
  <!-- Windows 8 -->
  <supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>
  <!-- Windows 8.1 -->
  <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>
  <!-- Windows 10 -->
  <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>
</application>
</compatibility>
</assembly>
```

Without the manifest file, the socket timeout will be overcorrected on these Windows Server versions and the actual time will be 500 milliseconds shorter than what was set with `GNAT.Sockets.Set_Socket_Option`. Note that on Microsoft Windows versions where correction is necessary, there is no way to set a socket timeout shorter than 500 ms. If a socket timeout shorter than 500 ms is needed on these Windows versions, you should add a call to `Check_Selector` before any socket read or write operations.

### 7.4.8 Mixed-Language Programming on Windows

Developing pure Ada applications on Windows is no different than on other GNAT-supported platforms. However, when developing or porting an application that contains a mix of Ada and C/C++, the choice of your Windows C/C++ development environment conditions your overall interoperability strategy.

If you use `gcc` or Microsoft C to compile the non-Ada part of your application, there are no Windows-specific restrictions that affect the overall interoperability with your Ada code. If you do want to use the Microsoft tools for your C++ code, you have two choices:

- \* You can encapsulate your C++ code in a DLL to be linked with your Ada application. In this case, use the Microsoft or other environment to build the DLL and use GNAT to build your executable ([Using DLLs with GNAT], page 252).

- \* You can encapsulate your Ada code in a DLL to be linked with the other part of your application. In this case, use GNAT to build the DLL ([Building DLLs with GNAT Project files], page 255) and use the Microsoft or other environment to build your executable.

In addition to the description about C `main` in [Mixed Language Programming], page 51, section, if the C `main` uses a stand-alone library, it is required on x86-windows to setup the SEH context. For this, the C `main` must look like this:

```
/* main.c */
extern void adainit (void);
extern void adafinal (void);
extern void __gnat_initialize(void*);
extern void call_to_ada (void);

int main (int argc, char *argv[])
{
    int SEH [2];

    /* Initialize the SEH context */
    __gnat_initialize (&SEH);

    adainit();

    /* Then call Ada services in the stand-alone library */

    call_to_ada();

    adafinal();
}
```

Note that you need not do this on x86\_64-windows where the Windows native SEH support is used.

#### 7.4.8.1 Windows Calling Conventions

This section pertains only to Win32. On Win64, there is a single native calling convention. All convention specifiers are ignored on this platform.

When a subprogram `F` (caller) calls a subprogram `G` (callee), there are several ways to push `G`'s parameters on the stack and there are several possible scenarios to clean up the stack upon `G`'s return. A calling convention is an agreed upon software protocol whereby the responsibilities between the caller (`F`) and the callee (`G`) are clearly defined. Several calling conventions are available for Windows:

- \* C (Microsoft defined)
- \* `Stdcall` (Microsoft defined)
- \* `Win32` (GNAT specific)
- \* `DLL` (GNAT specific)

### 7.4.8.2 C Calling Convention

This is the default calling convention used when interfacing to C/C++ routines compiled with either `gcc` or Microsoft Visual C++.

In the C calling convention, subprogram parameters are pushed on the stack by the caller from right to left. The caller itself is in charge of cleaning up the stack after the call. In addition, the name of a routine with C calling convention is mangled by adding a leading underscore.

The name to use on the Ada side when importing (or exporting) a routine with C calling convention is the name of the routine. For example you should import the C function:

```
int get_val (long);
```

from Ada as follows:

```
function Get_Val (V : Interfaces.C.long) return Interfaces.C.int;
pragma Import (C, Get_Val, External_Name => "get_val");
```

Note that in this particular case, you could have omitted the `External_Name` parameter since, when missing, this parameter is set to the name of the Ada entity in lower case. When the `Link_Name` parameter is missing, as in the above example, this parameter is set the `External_Name` with a leading underscore.

When importing a variable defined in C, you should always use the C calling convention unless the object containing the variable is part of a DLL (in which case you should use the `Stdcall` calling convention, [Stdcall Calling Convention], page 250).

### 7.4.8.3 Stdcall Calling Convention

This convention, which was the calling convention used for Pascal programs, is used by Microsoft for all the routines in the Win32 API for efficiency reasons. You must use it to import any routine for which this convention was specified.

In the `Stdcall` calling convention, subprogram parameters are also pushed on the stack by the caller from right to left. However, the callee, not the caller, is in charge of cleaning up the stack on routine exit. In addition, the name of a routine with `Stdcall` calling convention is mangled by adding a leading underscore (as for the C calling convention) and a trailing `@nn`, where `nn` is the overall size (in bytes) of the parameters passed to the routine.

The name to use on the Ada side when importing a C routine with a `Stdcall` calling convention is the name of the C routine. The leading underscore and trailing `@nn` are added automatically by the compiler. For example, you could import the Win32 function:

```
APIENTRY int get_val (long);
```

from Ada as follows:

```
function Get_Val (V : Interfaces.C.long) return Interfaces.C.int;
pragma Import (Stdcall, Get_Val);
-- On the x86 a long is 4 bytes, so the Link_Name is "_get_val@4"
```

Like the case for the C calling convention, when the `External_Name` parameter is missing, it is the name of the Ada entity in lower case. If instead of writing the above import pragma you write:

```
function Get_Val (V : Interfaces.C.long) return Interfaces.C.int;
pragma Import (Stdcall, Get_Val, External_Name => "retrieve_val");
```

the imported routine is `_retrieve_val@4`. However, if instead of specifying the `External_Name` parameter, you specify the `Link_Name` as in the following example:

```
function Get_Val (V : Interfaces.C.long) return Interfaces.C.int;
pragma Import (Stdcall, Get_Val, Link_Name => "retrieve_val");
```

the imported routine is `retrieve_val`. There is no decoration at all; no leading underscore and no `Stdcall` suffix `@nn`.

This is especially important as in some special cases a DLL's entry point name lacks a trailing `@nn` while the exported name generated for a call has it.

You can also import variables defined in a DLL by using an import pragma for a variable. As an example, if a DLL contains a variable defined as:

```
int my_var;
```

then, to access this variable from Ada you should write:

```
My_Var : Interfaces.C.int;
pragma Import (Stdcall, My_Var);
```

Note that to ease building cross-platform bindings, this convention will be handled as a C calling convention on non-Windows platforms.

#### 7.4.8.4 Win32 Calling Convention

This convention, which is GNAT-specific, is fully equivalent to the `Stdcall` calling convention described above.

#### 7.4.8.5 DLL Calling Convention

This convention, which is GNAT-specific, is fully equivalent to the `Stdcall` calling convention described above.

#### 7.4.8.6 Introduction to Dynamic Link Libraries (DLLs)

A Dynamically Linked Library (DLL) is a library that can be shared by several applications running under Windows. A DLL can contain any number of routines and variables.

One advantage of DLLs is that you can change and enhance them without forcing all the applications that depend on them to be relinked or recompiled. However, you should be aware that all calls to DLL routines are slower since, as you will understand below, such calls are indirect.

To illustrate the remainder of this section, suppose that an application wants to use the services of a DLL `API.dll`. To use the services provided by `API.dll`, you must statically link against the DLL or an import library which contains a jump table with an entry for each routine and variable exported by the DLL. In the Microsoft world, this import library is called `API.lib`. When using GNAT, this import library is called either `libAPI.dll.a`, `libapi.dll.a`, `libAPI.a` or `libapi.a` (names are case insensitive).

After you have linked your application with the DLL or the import library and you run your application, here is what happens:

- \* Your application is loaded into memory.
- \* The DLL `API.dll` is mapped into the address space of your application. This means that:
  - The DLL uses the stack of the calling thread.

- The DLL uses the virtual address space of the calling process.
- The DLL allocates memory from the virtual address space of the calling process.
- Handles (pointers) can be safely exchanged between routines in the DLL routines and routines in the application using the DLL.
- \* The entries in the jump table (from the import library `libAPI.dll.a` or `API.lib` or automatically created when linking against a DLL) which is part of your application are initialized with the addresses of the routines and variables in `API.dll`.
- \* If present in `API.dll`, routines `DllMain` or `DllMainCRTStartup` are invoked. These routines typically contain the initialization code needed for the well-being of the routines and variables exported by the DLL.

There is an additional point which is worth mentioning. In the Windows world, there are two kind of DLLs: relocatable and non-relocatable DLLs. Non-relocatable DLLs can only be loaded at a specific address in the target application address space. If the addresses of two non-relocatable DLLs overlap and these happen to be used by the same application, a conflict occurs and the application will run incorrectly. Hence, when possible, you should always use and build relocatable DLLs. Both relocatable and non-relocatable DLLs are supported by GNAT. Note that the `-s` linker switch (see GNU Linker User's Guide) removes the debugging symbols from the DLL, but the DLL can still be relocated.

As a side note, an interesting difference between Microsoft DLLs and Unix shared libraries is the fact that on most Unix systems all public routines are exported by default in a Unix shared library, while under Windows it is possible (but not required) to list exported routines in a definition file (see [The Definition File], page 253).

#### 7.4.8.7 Using DLLs with GNAT

To use the services of a DLL, say `API.dll`, in your Ada application you must have:

- \* The Ada spec for the routines and/or variables you want to access in `API.dll`. If not available, you must build this Ada spec from the C/C++ header files provided with the DLL.
- \* The import library (`libAPI.dll.a` or `API.lib`). As previously mentioned, an import library is a statically linked library containing the import table, which is filled at load time to point to the actual `API.dll` routines. Sometimes you don't have an import library for the DLL you want to use. The following sections will explain how to build one. Note that this is optional.
- \* The actual DLL, `API.dll`.

Once you have all the above, to compile an Ada application that uses the services of `API.dll` and whose main subprogram is `My_Ada_App`, you simply issue the command

```
$ gnatmake my_ada_app -larges -lAPI
```

The argument `-larges -lAPI` at the end of the `gnatmake` command tells the GNAT linker to look for an import library. The linker will look for a library name in this specific order:

- \* `libAPI.dll.a`
- \* `API.dll.a`
- \* `libAPI.a`
- \* `API.lib`

```
* libAPI.dll
* API.dll
```

The first three are the GNU-style import libraries. The third is the Microsoft-style import libraries. The last two are the actual DLL names.

Note that if the Ada package spec for `API.dll` contains the following pragma

```
pragma Linker_Options ("-lAPI");
```

you do not have to add `-largs -lAPI` at the end of the `gnatmake` command.

If any one of the items above is missing, you will have to create it yourself. The following sections explain how to do so using as an example a fictitious DLL called `API.dll`.

#### 7.4.8.8 Creating an Ada Spec for the DLL Services

A DLL typically comes with a C/C++ header file which provides the definitions of the routines and variables exported by the DLL. The Ada equivalent of this header file is a package spec that contains definitions for the imported entities. If the DLL you intend to use does not come with an Ada spec, you have to generate such a spec yourself. For example, if the header file of `API.dll` is a file `api.h` containing the following two definitions:

```
int some_var;
int get (char *);
```

then the equivalent Ada spec could be:

```
with Interfaces.C.Strings;
package API is
  use Interfaces;

  Some_Var : C.int;
  function Get (Str : C.Strings.Chars_Ptr) return C.int;

private
  pragma Import (C, Get);
  pragma Import (DLL, Some_Var);
end API;
```

#### 7.4.8.9 Creating an Import Library

If a Microsoft-style import library `API.lib` or a GNAT-style import library `libAPI.dll.a` or `libAPI.a` is available with `API.dll` you can skip this section. You can also skip this section if `API.dll` or `libAPI.dll` is built with GNU tools as in this case it is possible to link directly against the DLL. Otherwise read on.

#### The Definition File

As previously mentioned, and unlike Unix systems, the list of symbols that are exported from a DLL must be provided explicitly in Windows. The main goal of a definition file is precisely that: list the symbols exported by a DLL. A definition file (usually a file with a `.def` suffix) has the following structure:

```
[LIBRARY ``name``]
[DESCRIPTION ``string``]
```

```
EXPORTS
    ``symbol1``
    ``symbol2``
    ...
```

‘LIBRARY name’

This section, which is optional, gives the name of the DLL.

‘DESCRIPTION string’

This section, which is optional, gives a description string that will be embedded in the import library.

‘EXPORTS’

This section gives the list of exported symbols (procedures, functions or variables). For example, in the case of `API.dll` the `EXPORTS` section of `API.def` looks like:

```
EXPORTS
    some_var
    get
```

Note that you must specify the correct suffix (`@nn`) (see [Windows Calling Conventions], page 249) for a `Stdcall` calling convention function in the exported symbols list.

There can actually be other sections in a definition file, but these sections are not relevant to the discussion at hand.

## Creating a Definition File Automatically

You can automatically create the definition file `API.def` (see [The Definition File], page 253) from a DLL. To do that, use the `dlltool` program as follows:

```
$ dlltool API.dll -z API.def --export-all-symbols
```

Note that if some routines in the DLL have the `Stdcall` convention ([Windows Calling Conventions], page 249) with stripped `@nn` suffix then you’ll have to edit `api.def` to add it and specify `-k` to `gnatdll` when creating the import library.

Here are some hints to find the right `@nn` suffix.

- If you have the Microsoft import library (`.lib`), you may be able to find the right symbols by using the Microsoft `dumpbin` tool (see the corresponding Microsoft documentation for further details).

```
$ dumpbin /exports api.lib
```

- If you get a message about a missing symbol at link time, the compiler tells you what symbol is expected. You then can go back to the definition file and add the right suffix.

## GNAT-Style Import Library

To create a static import library from `API.dll` with the GNAT tools, you should create the `.def` file and use the `gnatdll` tool (see [Using gnatdll], page 260) as follows:

```
$ gnatdll -e API.def -d API.dll
```

`gnatdll` takes as input a definition file `API.def` and the name of the DLL containing the services listed in the definition file `API.dll`. The name of the static import library generated is computed from the name of the definition file as follows: if the definition file name is `xyz.def`, the import library name will be `libxyz.a`. Note that in the previous example, the switch `-e` could have been removed because the name of the definition file (before the `.def` suffix) is the same as the name of the DLL ([Using `gnatdll`], page 260, for more information about `gnatdll`).

## Microsoft-Style Import Library

A Microsoft import library is needed only if you plan to make an Ada DLL available to applications developed with Microsoft tools ([Mixed-Language Programming on Windows], page 248).

To create a Microsoft-style import library for `API.dll` you should create the `.def` file, then build the actual import library using Microsoft's `lib` utility:

```
$ lib -machine:IX86 -def:API.def -out:API.lib
```

If you use the above command, the definition file `API.def` must contain a line giving the name of the DLL:

```
LIBRARY      "API"
```

See the Microsoft documentation for further details about the usage of `lib`.

### 7.4.8.10 Building DLLs with GNAT Project files

There is nothing specific to Windows in the build process. See the ‘Library Projects’ section in the ‘GNAT Project Manager’ chapter of the ‘GPRbuild User’s Guide’.

Due to a system limitation, you cannot create threads under Windows when inside the `DllMain` routine which is used for auto-initialization of shared libraries, so you can’t have library level tasks in SALs.

### 7.4.8.11 Building DLLs with GNAT

This section explains how to build DLLs using the GNAT built-in DLL support. With the following procedure, it is straightforward to build and use DLLs with GNAT.

- \* Building object files. The first step is to build all objects files that are to be included into the DLL. This is done using the standard `gnatmake` tool.
- \* Building the DLL. To build the DLL, you must use the `gcc -shared` and `-shared-libgcc` switches. It’s quite simple to use this method:

```
$ gcc -shared -shared-libgcc -o api.dll obj1.o obj2.o ...
```

It’s important to note that in this case all symbols found in the object files are automatically exported. You can restrict the set of symbols to export by passing to `gcc` a definition file (see [The Definition File], page 253). For example:

```
$ gcc -shared -shared-libgcc -o api.dll api.def obj1.o obj2.o ...
```

If you use a definition file, you must export the elaboration procedures for every package that requires one. Elaboration procedures are named using the package name followed by “\_E”.

- \* Preparing DLL to be used. For the DLL to be used by client programs, the bodies must be hidden from it and the `.ali` set with read-only attribute. This is very important because otherwise GNAT will recompile all packages and will not actually use the code in the DLL. For example:

```
$ mkdir apilib
$ copy *.ads *.ali api.dll apilib
$ attrib +R apilib\*.ali
```

At this point, you can use the DLL by directly linking against it. Note that you must use the GNAT shared runtime when using GNAT shared libraries. You do this with the `-shared` binder switch.

```
$ gnatmake main -Iapilib -bargs -shared -largS -Lapilib -lAPI
```

#### 7.4.8.12 Building DLLs with gnatdll

Note that it is preferred to use GNAT Project files ([Building DLLs with GNAT Project files], page 255) or the built-in GNAT DLL support ([Building DLLs with GNAT], page 255) to build DLLs.

This section explains how to build DLLs containing Ada code using `gnatdll`. These DLLs will be referred to as Ada DLLs in the remainder of this section.

The steps required to build an Ada DLL that is to be used by Ada as well as non-Ada applications are as follows:

- \* You need to mark each Ada entity exported by the DLL with a `C` or `Stdcall` calling convention to avoid any Ada name mangling for the entities exported by the DLL (see [Exporting Ada Entities], page 257). You can skip this step if you plan to use the Ada DLL only from Ada applications.
- \* Your Ada code must export an initialization routine which calls the routine `adainit` (generated by `gnatbind`) to perform the elaboration of the Ada code in the DLL ([Ada DLLs and Elaboration], page 258). The initialization routine exported by the Ada DLL must be invoked by the clients of the DLL to initialize the DLL.
- \* When useful, the DLL should also export a finalization routine which calls routine `adafinal` (also generated by `gnatbind`) to perform the finalization of the Ada code in the DLL ([Ada DLLs and Finalization], page 259). The finalization routine exported by the Ada DLL must be invoked by the clients of the DLL when the DLL services are no further needed.
- \* You must provide a spec for the services exported by the Ada DLL in each of the programming languages to which you plan to make the DLL available.
- \* You must provide a definition file listing the exported entities ([The Definition File], page 253).
- \* Finally, you must use `gnatdll` to produce the DLL and the import library ([Using gnatdll], page 260).

Note that a relocatable DLL stripped using the `strip` binutils tool is no longer relocatable. To build a DLL without debug information, pass `-largS -s` to `gnatdll`. This restriction does not apply to a DLL built using a Library Project. See the ‘Library Projects’ section in the ‘GNAT Project Manager’ chapter of the ‘GPRbuild User’s Guide’.

### 7.4.8.13 Limitations When Using Ada DLLs from Ada

When using Ada DLLs from Ada applications there is a limitation you should be aware of. On Windows, the GNAT run-time is not in a DLL of its own, so each Ada DLL includes a part of the GNAT run-time. Specifically, each Ada DLL includes the services of the GNAT run-time that are necessary for the Ada code inside the DLL. As a result, when an Ada program uses an Ada DLL there are two independent GNAT run-times: one in the Ada DLL and one in the main program.

It is therefore not possible to exchange GNAT run-time objects between the Ada DLL and the main Ada program. Example of GNAT run-time objects are file handles (e.g., `Text_IO.File_Type`), tasks types, protected objects types, etc.

It is completely safe to exchange plain elementary, array or record types, Windows object handles, etc.

### 7.4.8.14 Exporting Ada Entities

Building a DLL is a way to encapsulate a set of services usable from any application. As a result, the Ada entities exported by a DLL should be exported with the `C` or `Stdcall` calling conventions to avoid any Ada name mangling. As an example here is an Ada package `API`, spec and body, exporting two procedures, a function, and a variable:

```
with Interfaces.C; use Interfaces;
package API is
  Count : C.int := 0;
  function Factorial (Val : C.int) return C.int;

  procedure Initialize_API;
  procedure Finalize_API;
  -- Initialization & Finalization routines. More in the next section.
private
  pragma Export (C, Initialize_API);
  pragma Export (C, Finalize_API);
  pragma Export (C, Count);
  pragma Export (C, Factorial);
end API;

package body API is
  function Factorial (Val : C.int) return C.int is
    Fact : C.int := 1;
  begin
    Count := Count + 1;
    for K in 1 .. Val loop
      Fact := Fact * K;
    end loop;
    return Fact;
  end Factorial;

  procedure Initialize_API is
    procedure Adainit;
```

```

        pragma Import (C, Adainit);
    begin
        Adainit;
    end Initialize_API;

    procedure Finalize_API is
        procedure Adafinal;
        pragma Import (C, Adafinal);
    begin
        Adafinal;
    end Finalize_API;
end API;

```

If the Ada DLL you are building will only be used by Ada applications, you do not have to export Ada entities with a C or Stdcall convention. As an example, the previous package could be written as follows:

```

package API is
    Count : Integer := 0;
    function Factorial (Val : Integer) return Integer;

    procedure Initialize_API;
    procedure Finalize_API;
    -- Initialization and Finalization routines.
end API;

package body API is
    function Factorial (Val : Integer) return Integer is
        Fact : Integer := 1;
    begin
        Count := Count + 1;
        for K in 1 .. Val loop
            Fact := Fact * K;
        end loop;
        return Fact;
    end Factorial;

    ...
    -- The remainder of this package body is unchanged.
end API;

```

Note that if you do not export the Ada entities with a C or Stdcall convention, you will have to provide the mangled Ada names in the definition file of the Ada DLL ([Creating the Definition File], page 260).

#### 7.4.8.15 Ada DLLs and Elaboration

The DLL that you are building contains your Ada code as well as all the routines in the Ada library that are needed by it. The first thing a user of your DLL must do is elaborate the Ada code ([Elaboration Order Handling in GNAT], page 286).

To allow this, you must export an initialization routine (`Initialize_Api` in the previous example), which must be invoked before using any of the DLL services. This elaboration routine must call the Ada elaboration routine `adainit` generated by the GNAT binder ([Binding with Non-Ada Main Programs], page 166). See the body of `Initialize_Api` for an example. Note that the GNAT binder is automatically invoked during the DLL build process by the `gnatdll` tool ([Using gnatdll], page 260).

When a DLL is loaded, Windows systematically invokes a routine called `DllMain`. It should therefore be possible to call `adainit` directly from `DllMain` without having to provide an explicit initialization routine. Unfortunately, you can't call `adainit` from the `DllMain` if your program has library level tasks because access to the `DllMain` entry point is serialized by the system (that is, only a single thread can execute 'through' it at a time), which means that the GNAT run-time will deadlock waiting for a newly created task to complete its initialization.

#### 7.4.8.16 Ada DLLs and Finalization

When the services of an Ada DLL are no longer needed, the client code should invoke the DLL finalization routine, if available. The DLL finalization routine is in charge of releasing all resources acquired by the DLL. In the case of the Ada code contained in the DLL, this is achieved by calling routine `adafinal` generated by the GNAT binder ([Binding with Non-Ada Main Programs], page 166). See the body of `Finalize_Api` for an example. As already pointed out the GNAT binder is automatically invoked during the DLL build process by the `gnatdll` tool ([Using gnatdll], page 260).

#### 7.4.8.17 Creating a Spec for Ada DLLs

To use the services exported by the Ada DLL from another programming language (e.g., C), you have to translate the specs of the exported Ada entities in that language. For instance in the case of `API.dll`, the corresponding C header file could look like:

```
extern int *_imp__count;
#define count (*_imp__count)
int factorial (int);
```

It is important to understand that when building an Ada DLL to be used by other Ada applications, you need two different specs for the packages contained in the DLL: one for building the DLL and the other for using the DLL. This is because the DLL calling convention is needed to use a variable defined in a DLL, but when building the DLL, the variable must have either the **Ada** or **C** calling convention. As an example consider a DLL consisting of the following package `API`:

```
package API is
  Count : Integer := 0;
  ...
  -- Remainder of the package omitted.
end API;
```

After producing a DLL containing package `API`, the spec that must be used to import `API.Count` from Ada code outside of the DLL is:

```
package API is
  Count : Integer;
```

```

        pragma Import (DLL, Count);
    end API;

```

#### 7.4.8.18 Creating the Definition File

The definition file is the last file you need to build the DLL. It lists the exported symbols. As an example, the definition file for a DLL containing only package `API` above (where all the entities are exported with a C calling convention) is:

```

EXPORTS
    count
    factorial
    finalize_api
    initialize_api

```

If the C calling convention is missing from package `API`, the definition file contains the mangled Ada names of the above entities, which in this case are:

```

EXPORTS
    api__count
    api__factorial
    api__finalize_api
    api__initialize_api

```

#### 7.4.8.19 Using `gnatdll`

`gnatdll` is a tool to automate the DLL build process once all the Ada and non-Ada sources that make up your DLL have been compiled. `gnatdll` is actually in charge of two distinct tasks: building both the static import library for the DLL and the actual DLL. You invoke the `gnatdll` command as

```
$ gnatdll [ switches ] list-of-files [ -larges opts ]
```

where `list-of-files` is a list of ALI and object files. The object file list must be the exact list of objects corresponding to the non-Ada sources whose services are to be included in the DLL. The ALI file list must be the exact list of ALI files for the corresponding Ada sources whose services are to be included in the DLL. If `list-of-files` is missing, only the static import library is generated.

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

`-a[`address']`

Build a non-relocatable DLL at `address`. If you don't specify `address`, `gnatdll` uses the default address of `0x11000000`. By default, when this switch is missing, `gnatdll` builds a relocatable DLL. We advise you to build relocatable DLL.

`-bargs `opts'`

Binder switches. Pass `opts` to the binder.

`-d `dllfile'`

`dllfile` is the name of the DLL. You must specify this switch for `gnatdll` to do anything. `gnatdll` names the generated import library algorithmically from `dllfile` as shown in the following example: if `dllfile` is `xyz.dll`, the import library name is `libxyz.dll.a`. `gnatdll` obtains the name of the definition file (if not specified by switch `-e`) algorithmically from `dllfile` as shown in the following example: if `dllfile` is `xyz.dll`, the definition file used is `xyz.def`.

- e ``deffile'`**  
`deffile` is the name of the definition file.
- g**  
Generate debugging information. This information is stored in the object file and copied from there to the final DLL 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 or the symbolic stack traceback.
- h**  
Help mode. Displays `gnatdll` switch usage information.
- I``dir'`**  
Direct `gnatdll` to search the `dir` directory for source and object files needed to build the DLL. ([Search Paths and the Run-Time Library (RTL)], page 89).
- k**  
Removes the `@nn` suffix from the import library's exported names, but keeps them for the link names. You must specify this switch if you want to use a `Stdcall` function in a DLL for which the `@nn` suffix has been removed. This is the case for most of the Windows NT DLL for example. This switch has no effect if you specify the `-n` switch.
- l ``file'`**  
The list of ALI and object files used to build the DLL are listed in `file`, instead of being given in the command line. Each line in `file` contains the name of an ALI or object file.
- n**  
No Import. Do not create the import library.
- q**  
Quiet mode. Do not display unnecessary messages.
- v**  
Verbose mode. Display extra information.
- larges ``opts'`**  
Linker switches. Pass `opts` to the linker.

### gnatdll Example

As an example, the command to build a relocatable DLL from `api.adb` once `api.adb` has been compiled and `api.def` created is

```
$ gnatdll -d api.dll api.ali
```

The above command creates two files: `libapi.dll.a` (the import library) and `api.dll` (the actual DLL). If you want to create only the DLL, just type:

```
$ gnatdll -d api.dll -n api.ali
```

Alternatively, if you want to create just the import library, type:

```
$ gnatdll -d api.dll
```

## gnatdll behind the Scenes

This section details the steps involved in creating a DLL. `gnatdll` does these steps for you. Unless you are interested in understanding what goes on behind the scenes, you should skip this section.

We use the previous example of a DLL containing the Ada package `API`, to illustrate the steps necessary to build a DLL. The starting point is a set of objects that make up the DLL and the corresponding ALI files. In the case of this example, this means `api.o` and `api.ali`. To build a relocatable DLL, `gnatdll` does the following:

- \* builds the base file (`api.base`). A base file gives the information necessary to generate relocation information for the DLL.

```
$ gnatbind -n api
$ gnatlink api -o api.jnk -mdll -Wl,--base-file,api.base
```

In addition to the base file, the `gnatlink` command generates an output file `api.jnk`, which can be discarded. The `-mdll` switch asks `gnatlink` to generate the routines `DllMain` and `DllMainCRTStartup` that are called by the Windows loader when the DLL is loaded into memory.

- \* uses `dlltool` (see [Using `dlltool`], page 262) to build the export table (`api.exp`). The export table contains the relocation information in a form which can be used during the final link to ensure that the Windows loader is able to place the DLL anywhere in memory.

```
$ dlltool --dllname api.dll --def api.def --base-file api.base \
--output-exp api.exp
```

- \* builds the base file using the new export table. Note that `gnatbind` must be called once again since the binder generated file has been deleted during the previous call to `gnatlink`.

```
$ gnatbind -n api
$ gnatlink api -o api.jnk api.exp -mdll
-Wl,--base-file,api.base
```

- \* builds the new export table using the new base file and generates the DLL import library `libAPI.dll.a`.

```
$ dlltool --dllname api.dll --def api.def --base-file api.base \
--output-exp api.exp --output-lib libAPI.a
```

- \* Finally, builds the relocatable DLL using the final export table.

```
$ gnatbind -n api
$ gnatlink api api.exp -o api.dll -mdll
```

## Using `dlltool`

`dlltool` is the low-level tool used by `gnatdll` to build DLLs and static import libraries. This section summarizes the most common `dlltool` switches. You run `dlltool` as follows:

```
$ dlltool [`switches`]
```

`dlltool` switches include:

```
--base-file `basefile`
```

Read the base file `basefile` generated by the linker. You use this switch to create a relocatable DLL.

```

--def `deffile'
    Read the definition file.

--dllname `name'
    Gives the name of the DLL. You use this switch to embed the name of the DLL
    in the static import library generated by dlltool with switch --output-lib.

-k
    Kill @nn from exported names ([Windows Calling Conventions], page 249, for a
    discussion about Stdcall-style symbols).

--help
    Prints the dlltool switches with a concise description.

--output-exp `exportfile'
    Generate an export file exportfile. The export file contains the export table
    (list of symbols in the DLL) and is used to create the DLL.

--output-lib `libfile'
    Generate a static import library libfile.

-v
    Verbose mode.

--as `assembler-name'
    Use assembler-name as the assembler. The default is as.

```

#### 7.4.8.20 GNAT and Windows Resources

Resources are an easy way to add Windows-specific objects to your application. The objects that you can add as resources include:

- \* menus
- \* accelerators
- \* dialog boxes
- \* string tables
- \* bitmaps
- \* cursors
- \* icons
- \* fonts
- \* version information

You can use a version information resource to embed information into an executable or a DLL. This information can be viewed using the file properties from the Windows Explorer. Here's an example of a version information resource:

```

1 VERSIONINFO
FILEVERSION      1,0,0,0
PRODUCTVERSION  1,0,0,0
BEGIN
    BLOCK "StringFileInfo"

```

```

BEGIN
  BLOCK "080904E4"
  BEGIN
    VALUE "CompanyName", "My Company Name"
    VALUE "FileDescription", "My application"
    VALUE "FileVersion", "1.0"
    VALUE "InternalName", "my_app"
    VALUE "LegalCopyright", "My Name"
    VALUE "OriginalFilename", "my_app.exe"
    VALUE "ProductName", "My App"
    VALUE "ProductVersion", "1.0"
  END
END

  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x809, 1252
  END
END

```

The value 0809 (langID) is for the U.K English language and 04E4 (charsetID), which is equal to 1252 decimal, for multilingual.

This section explains how to build, compile and use resources. Note that this section does not cover all resource objects; for a complete description see the corresponding Microsoft documentation.

#### 7.4.8.21 Building Resources

A resource file is an ASCII file. By convention, resource files have an `.rc` extension. The easiest way to build a resource file is to use Microsoft tools such as `imagedit.exe` to build bitmaps, icons and cursors and `dlgedit.exe` to build dialogs. You can always build an `.rc` file yourself by writing a resource script.

It's not our objective to explain how to write a resource file. A complete description of the resource script language can be found in the Microsoft documentation.

#### 7.4.8.22 Compiling Resources

This section describes how you can build a GNAT-compatible (COFF) object file containing the resources. You do this using the Resource Compiler `windres` as follows:

```
$ windres -i myres.rc -o myres.o
```

By default `windres` runs `gcc` to preprocess the `.rc` file. You can specify an alternate preprocessor (usually named `cpp.exe`) using the `windres --preprocessor` parameter. You can obtain a list of all possible switches by entering the command `windres --help`.

You can also use the Microsoft resource compiler `rc.exe` to produce a `.res` file (binary resource file). See the corresponding Microsoft documentation for further details. In this case, you need to use `windres` to translate the `.res` file to a GNAT-compatible object file as follows:

```
$ windres -i myres.res -o myres.o
```

### 7.4.8.23 Using Resources

To include the resource file in your program just add the GNAT-compatible object file for the resource(s) to the linker arguments. With `gnatmake` you do this using the `-largrs` switch:

```
$ gnatmake myprog -largrs myres.o
```

### 7.4.8.24 Using GNAT DLLs from Microsoft Visual Studio Applications

This section describes a common case of mixed GNAT/Microsoft Visual Studio application development, where the main program is developed using MSVS and is linked with a DLL developed using GNAT. You should develop such a mixed application following the general guidelines outlined above; below is the cookbook-style sequence of steps to follow:

1. First develop and build the GNAT shared library using a library project (let's assume the project is `mylib.gpr`, producing the library `libmylib.dll`):

```
$ gprbuild -p mylib.gpr
```

2. Produce a `.def` file for the symbols you need to interface with, either by hand or automatically with possibly some manual adjustments (see [Creating Definition File Automatically], page 254):

```
$ dlltool libmylib.dll -z libmylib.def --export-all-symbols
```

3. Make sure that MSVS command-line tools are accessible on the path.
4. Create the Microsoft-style import library (see [MSVS-Style Import Library], page 255):

```
$ lib -machine:IX86 -def:libmylib.def -out:libmylib.lib
```

If you are using a 64-bit toolchain, the above becomes...

```
$ lib -machine:X64 -def:libmylib.def -out:libmylib.lib
```

5. Build the C main:

```
$ cl /O2 /MD main.c libmylib.lib
```

6. Before running the executable, make sure you have set the `PATH` to include the DLL or copy the DLL into the directory containing the `.exe`.

### 7.4.8.25 Debugging a DLL

Debugging a DLL is similar to debugging a standard program, but you have to deal with two different executable parts: the DLL and the program that uses it. There are the following four possibilities:

- \* The program and DLL are built with GCC/GNAT.
- \* The program is built with foreign tools and the DLL is built with GCC/GNAT.
- \* The program is built with GCC/GNAT and the DLL is built with foreign tools.

In this section we address only cases one and two above. Note that there is no point in trying to debug a DLL with GNU/GDB if there is no GDB-compatible debugging information in it. To do so, you must use a debugger compatible with the tools suite used to build the DLL.

#### 7.4.8.26 Program and DLL Both Built with GCC/GNAT

This is the simplest case. Both the DLL and the program have GDB compatible debugging information. You can then break anywhere in the process. Let's suppose the main procedure is named `ada_main` and in the DLL there's an entry point named `ada_dll`.

The DLL ([Introduction to Dynamic Link Libraries (DLLs)], page 251) and program must have been built with the debugging information (see the GNAT `-g` switch). Here are the step-by-step instructions for debugging it:

- \* Launch GDB on the main program.

```
$ gdb -nw ada_main
```

- \* Start the program and stop at the beginning of the main procedure

```
(gdb) start
```

This step is required to be able to set a breakpoint inside the DLL. Until the program is run, the DLL is not loaded. This has the consequence that the DLL debugging information is also not loaded, so it is not possible to set a breakpoint in the DLL.

- \* Set a breakpoint inside the DLL

```
(gdb) break ada_dll
```

```
(gdb) cont
```

At this stage, a breakpoint is set inside the DLL. From there on you can use standard GDB commands to debug the whole program ([Running and Debugging Ada Programs], page 184).

#### 7.4.8.27 Program Built with Foreign Tools and DLL Built with GCC/GNAT

In this case, things are slightly more complex because you can't start the main program and then break at the beginning to load the DLL and the associated DLL debugging information. It's not possible to break at the beginning of the program because there's no GDB debugging information, and therefore there's no direct way of getting initial control. This section addresses this issue by describing some methods that you can use to break somewhere in the DLL to debug it.

First, suppose that the main procedure is named `main` (this is the case, for example, for some C code built with Microsoft Visual C) and that there's a DLL named `test.dll` containing an Ada entry point named `ada_dll`.

The DLL (see [Introduction to Dynamic Link Libraries (DLLs)], page 251) must have been built with debugging information (see the GNAT `-g` switch).

#### Debugging the DLL Directly

- \* Determine the executable's starting address

```
$ objdump --file-header main.exe
```

The starting address is reported on the last line. For example:

```
main.exe:      file format pei-i386
architecture: i386, flags 0x0000010a:
EXEC_P, HAS_DEBUG, D_PAGED
start address 0x00401010
```

- \* Launch the debugger on the executable.

```
$ gdb main.exe
```

- \* Set a breakpoint at the starting address and launch the program.

```
$ (gdb) break *0x00401010
$ (gdb) run
```

The program will stop at the specified address.

- \* Set a breakpoint on a DLL subroutine.

```
(gdb) break ada_dll.adb:45
```

Or if you want to break using a symbol on the DLL, you need first to select the Ada language (language used by the DLL).

```
(gdb) set language ada
(gdb) break ada_dll
```

- \* Continue the program.

```
(gdb) cont
```

This runs the program until it reaches the breakpoint that you've set. From that point, you can use standard GDB commands to debug a program as described in ([Running and Debugging Ada Programs], page 184).

You can also debug the DLL by attaching GDB to a running process.

## Attaching to a Running Process

With GDB, you can always debug a running process by attaching to it. It's possible to debug a DLL this way. The limitation of this approach is that the DLL must run long enough to perform the attach operation. To ensure this, you may want, for example, to insert a time-wasting loop in the code of the DLL to allow this to happen.

- \* Launch the main program `main.exe`.

```
$ main
```

- \* Use the Windows 'Task Manager' to find the process ID. Let's say that the process PID for `main.exe` is 208.

- \* Launch gdb.

```
$ gdb
```

- \* Attach to the running process to be debugged.

```
(gdb) attach 208
```

- \* Load the process debugging information.

```
(gdb) symbol-file main.exe
```

- \* Break somewhere in the DLL.

```
(gdb) break ada_dll
```

- \* Continue process execution.

```
(gdb) cont
```

This last step will resume the process execution and stop at the breakpoint we have set. From there you can use standard GDB commands to debug a program, as described in [Running and Debugging Ada Programs], page 184.

#### 7.4.8.28 Setting Stack Size from gnatlink

You can specify the program stack size at link time. On most versions of Windows, starting with XP, this is mostly useful to set the size of the main stack (environment task). The other task stacks are set with `pragma Storage_Size` or with the ‘`gnatbind -d`’ command. The specified size will become the reserved memory size of the underlying thread.

Since very old versions of Windows (2000, NT4, etc.) don’t allow setting the reserve size of individual tasks, for those versions the link-time stack size applies to all tasks, and `pragma Storage_Size` has no effect. In particular, Stack Overflow checks are made against this link-time specified size.

You can set this with `gnatlink` using either of the following:

- \* `-Xlinker` linker switch

```
$ gnatlink hello -Xlinker --stack=0x10000,0x1000
```

This sets the stack reserve size to 0x10000 bytes and the stack commit size to 0x1000 bytes.

- \* `-Wl` linker switch

```
$ gnatlink hello -Wl,--stack=0x1000000
```

This sets the stack reserve size to 0x1000000 bytes. Note that with `-Wl` switch, you can’t also set the stack commit size because the comma is a separator for this switch.

#### 7.4.8.29 Setting Heap Size from gnatlink

Under Windows systems, it is possible to specify the program heap size from `gnatlink` using either of the following:

- \* `-Xlinker` linker switch

```
$ gnatlink hello -Xlinker --heap=0x10000,0x1000
```

This sets the heap reserve size to 0x10000 bytes and the heap commit size to 0x1000 bytes.

- \* `-Wl` linker switch

```
$ gnatlink hello -Wl,--heap=0x1000000
```

This sets the heap reserve size to 0x1000000 bytes. Note that with `-Wl` switch, you can’t also set the heap commit size because the comma is a separator for this switch.

### 7.4.9 Windows Specific Add-Ons

This section describes the Windows specific add-ons.

#### 7.4.9.1 Win32Ada

`Win32Ada` is a binding for the Microsoft Win32 API, which you can easily install using the provided installer. To use it, you need to use a project file and add a single `with`-clause to give you full access to the `Win32Ada` binding sources and ensure that the proper libraries are passed to the linker.

```
with "win32ada";
project P is
  for Sources use ...;
end P;
```

To build the application, you just need to call `gprbuild` for the application's project, here `p.gpr`:

```
gprbuild p.gpr
```

### 7.4.9.2 wPOSIX

`wPOSIX` is a minimal POSIX binding whose goal is to help with building cross-platforms applications. This binding is not complete though, as the Win32 API does not provide the necessary support for all POSIX APIs.

To use the `wPOSIX` binding, you need to use a project file and add a single 'with' clause to give you full access to the `wPOSIX` binding sources and ensure that the proper libraries are passed to the linker.

```
with "wposix";
project P is
  for Sources use ...;
end P;
```

To build the application, you just need to call `gprbuild` for the application's project, here `p.gpr`:

```
gprbuild p.gpr
```

## 7.5 Mac OS Topics

This section describes topics that are specific to Apple's OS X platform.

### 7.5.1 Codesigning the Debugger

The Darwin Kernel, used by Apple's OS X, requires the debugger to have special permissions before it's allowed to control other processes. These permissions are granted by codesigning the GDB executable. Without these permissions, the debugger will report error messages such as:

```
Starting program: /x/y/foo
Unable to find Mach task port for process-id 28885: (os/kern) failure (0x5).
(please check gdb is codesigned - see taskgated(8))
```

Codesigning requires a certificate. The following procedure explains how to create one:

- \* Start the Keychain Access application (in `/Applications/Utilities/Keychain Access.app`)
- \* Select the Keychain Access -> Certificate Assistant -> Create a Certificate... menu
- \* Then:
  - \* Choose a name for the new certificate (this procedure will use "gdb-cert" as an example)
  - \* Set "Identity Type" to "Self Signed Root"
  - \* Set "Certificate Type" to "Code Signing"
  - \* Activate the "Let me override defaults" option
- \* Click several times on "Continue" until the "Specify a Location For The Certificate" screen appears, then set "Keychain" to "System"

- \* Click on “Continue” until the certificate is created
- \* Finally, in the view, double-click on the new certificate, and set “When using this certificate” to “Always Trust”
- \* Exit the Keychain Access application and restart the computer (this is unfortunately required)

Once you’ve created a certificate as above, you can codesign the debugger by running the following command in a Terminal:

```
$ codesign -f -s "gdb-cert" <gnat_install_prefix>/bin/gdb
```

with `gdb-cert` replaced by the actual certificate name chosen above, and `gnat_install_prefix` replaced by the location where you installed GNAT. Also, be sure that users of GDB are in the Unix group `_developer`.

## 8 Example of Binder Output File

This Appendix displays the source code for the output file generated by `gnatbind` for a simple ‘Hello World’ program. Comments have been added for clarification purposes.

```
-- The package is called Ada_Main unless this name is actually used
-- as a unit name in the partition, in which case some other unique
-- name is used.

pragma Ada_95;
with System;
package ada_main is
  pragma Warnings (Off);

  -- The main program saves the parameters (argument count,
  -- argument values, environment pointer) in global variables
  -- for later access by other units including Ada.Command_Line.

  gnat_argc : Integer;
  gnat_argv : System.Address;
  gnat_envp : System.Address;

  -- The actual variables are stored in a library routine. This
  -- is useful for some shared library situations, where there
  -- are problems if variables are not in the library.

  pragma Import (C, gnat_argc);
  pragma Import (C, gnat_argv);
  pragma Import (C, gnat_envp);

  -- The exit status is similarly an external location

  gnat_exit_status : Integer;
  pragma Import (C, gnat_exit_status);

  GNAT_Version : constant String :=
    "GNAT Version: Pro 7.4.0w (20141119-49)" & ASCII.NUL;
  pragma Export (C, GNAT_Version, "__gnat_version");

  Ada_Main_Program_Name : constant String := "_ada_hello" & ASCII.NUL;
  pragma Export (C, Ada_Main_Program_Name, "__gnat_ada_main_program_name");

  -- This is the generated adainit routine that performs
  -- initialization at the start of execution. In the case
  -- where Ada is the main program, this main program makes
  -- a call to adainit at program startup.

  procedure adainit;
```

```

pragma Export (C, adainit, "adainit");

-- This is the generated adafinal routine that performs
-- finalization at the end of execution. In the case where
-- Ada is the main program, this main program makes a call
-- to adafinal at program termination.

procedure adafinal;
pragma Export (C, adafinal, "adafinal");

-- This routine is called at the start of execution. It is
-- a dummy routine that is used by the debugger to breakpoint
-- at the start of execution.

-- This is the actual generated main program (it would be
-- suppressed if the no main program switch were used). As
-- required by standard system conventions, this program has
-- the external name main.

function main
  (argc : Integer;
   argv : System.Address;
   envp : System.Address)
  return Integer;
pragma Export (C, main, "main");

-- The following set of constants give the version
-- identification values for every unit in the bound
-- partition. This identification is computed from all
-- dependent semantic units, and corresponds to the
-- string that would be returned by use of the
-- Body_Version or Version attributes.

-- The following Export pragmas export the version numbers
-- with symbolic names ending in B (for body) or S
-- (for spec) so that they can be located in a link. The
-- information provided here is sufficient to track down
-- the exact versions of units used in a given build.

type Version_32 is mod 2 ** 32;
u00001 : constant Version_32 := 16#8ad6e54a#;
pragma Export (C, u00001, "helloB");
u00002 : constant Version_32 := 16#fbff4c67#;
pragma Export (C, u00002, "system__standard_libraryB");
u00003 : constant Version_32 := 16#1ec6fd90#;
pragma Export (C, u00003, "system__standard_libraryS");
u00004 : constant Version_32 := 16#3ffc8e18#;

```

```
pragma Export (C, u00004, "adaS");
u00005 : constant Version_32 := 16#28f088c2#;
pragma Export (C, u00005, "ada__text_ioB");
u00006 : constant Version_32 := 16#f372c8ac#;
pragma Export (C, u00006, "ada__text_ioS");
u00007 : constant Version_32 := 16#2c143749#;
pragma Export (C, u00007, "ada__exceptionsB");
u00008 : constant Version_32 := 16#f4f0cce8#;
pragma Export (C, u00008, "ada__exceptionsS");
u00009 : constant Version_32 := 16#a46739c0#;
pragma Export (C, u00009, "ada__exceptions__last_chance_handlerB");
u00010 : constant Version_32 := 16#3aac8c92#;
pragma Export (C, u00010, "ada__exceptions__last_chance_handlerS");
u00011 : constant Version_32 := 16#1d274481#;
pragma Export (C, u00011, "systemS");
u00012 : constant Version_32 := 16#a207fefe#;
pragma Export (C, u00012, "system__soft_linksB");
u00013 : constant Version_32 := 16#467d9556#;
pragma Export (C, u00013, "system__soft_linksS");
u00014 : constant Version_32 := 16#b01dad17#;
pragma Export (C, u00014, "system__parametersB");
u00015 : constant Version_32 := 16#630d49fe#;
pragma Export (C, u00015, "system__parametersS");
u00016 : constant Version_32 := 16#b19b6653#;
pragma Export (C, u00016, "system__secondary_stackB");
u00017 : constant Version_32 := 16#b6468be8#;
pragma Export (C, u00017, "system__secondary_stackS");
u00018 : constant Version_32 := 16#39a03df9#;
pragma Export (C, u00018, "system__storage_elementsB");
u00019 : constant Version_32 := 16#30e40e85#;
pragma Export (C, u00019, "system__storage_elementsS");
u00020 : constant Version_32 := 16#41837d1e#;
pragma Export (C, u00020, "system__stack_checkingB");
u00021 : constant Version_32 := 16#93982f69#;
pragma Export (C, u00021, "system__stack_checkingS");
u00022 : constant Version_32 := 16#393398c1#;
pragma Export (C, u00022, "system__exception_tableB");
u00023 : constant Version_32 := 16#b33e2294#;
pragma Export (C, u00023, "system__exception_tableS");
u00024 : constant Version_32 := 16#ce4af020#;
pragma Export (C, u00024, "system__exceptionsB");
u00025 : constant Version_32 := 16#75442977#;
pragma Export (C, u00025, "system__exceptionsS");
u00026 : constant Version_32 := 16#37d758f1#;
pragma Export (C, u00026, "system__exceptions__machineS");
u00027 : constant Version_32 := 16#b895431d#;
pragma Export (C, u00027, "system__exceptions_debugB");
```

```
u00028 : constant Version_32 := 16#aec55d3f#;
pragma Export (C, u00028, "system__exceptions_debugS");
u00029 : constant Version_32 := 16#570325c8#;
pragma Export (C, u00029, "system__img_intB");
u00030 : constant Version_32 := 16#1ffca443#;
pragma Export (C, u00030, "system__img_intS");
u00031 : constant Version_32 := 16#b98c3e16#;
pragma Export (C, u00031, "system__tracebackB");
u00032 : constant Version_32 := 16#831a9d5a#;
pragma Export (C, u00032, "system__tracebackS");
u00033 : constant Version_32 := 16#9ed49525#;
pragma Export (C, u00033, "system__traceback_entriesB");
u00034 : constant Version_32 := 16#1d7cb2f1#;
pragma Export (C, u00034, "system__traceback_entriesS");
u00035 : constant Version_32 := 16#8c33a517#;
pragma Export (C, u00035, "system__wch_conB");
u00036 : constant Version_32 := 16#065a6653#;
pragma Export (C, u00036, "system__wch_conS");
u00037 : constant Version_32 := 16#9721e840#;
pragma Export (C, u00037, "system__wch_stwB");
u00038 : constant Version_32 := 16#2b4b4a52#;
pragma Export (C, u00038, "system__wch_stwS");
u00039 : constant Version_32 := 16#92b797cb#;
pragma Export (C, u00039, "system__wch_cnvB");
u00040 : constant Version_32 := 16#09eddca0#;
pragma Export (C, u00040, "system__wch_cnvS");
u00041 : constant Version_32 := 16#6033a23f#;
pragma Export (C, u00041, "interfacesS");
u00042 : constant Version_32 := 16#ece6fdb6#;
pragma Export (C, u00042, "system__wch_jisB");
u00043 : constant Version_32 := 16#899dc581#;
pragma Export (C, u00043, "system__wch_jisS");
u00044 : constant Version_32 := 16#10558b11#;
pragma Export (C, u00044, "ada__streamsB");
u00045 : constant Version_32 := 16#2e6701ab#;
pragma Export (C, u00045, "ada__streamsS");
u00046 : constant Version_32 := 16#db5c917c#;
pragma Export (C, u00046, "ada__io_exceptionsS");
u00047 : constant Version_32 := 16#12c8cd7d#;
pragma Export (C, u00047, "ada__tagsB");
u00048 : constant Version_32 := 16#ce72c228#;
pragma Export (C, u00048, "ada__tagsS");
u00049 : constant Version_32 := 16#c3335bfd#;
pragma Export (C, u00049, "system__htableB");
u00050 : constant Version_32 := 16#99e5f76b#;
pragma Export (C, u00050, "system__htableS");
u00051 : constant Version_32 := 16#089f5cd0#;
```

```
pragma Export (C, u00051, "system__string_hashB");
u00052 : constant Version_32 := 16#3bbb9c15#;
pragma Export (C, u00052, "system__string_hashS");
u00053 : constant Version_32 := 16#807fe041#;
pragma Export (C, u00053, "system__unsigned_typesS");
u00054 : constant Version_32 := 16#d27be59e#;
pragma Export (C, u00054, "system__val_lluB");
u00055 : constant Version_32 := 16#fa8db733#;
pragma Export (C, u00055, "system__val_lluS");
u00056 : constant Version_32 := 16#27b600b2#;
pragma Export (C, u00056, "system__val_utilB");
u00057 : constant Version_32 := 16#b187f27f#;
pragma Export (C, u00057, "system__val_utilS");
u00058 : constant Version_32 := 16#d1060688#;
pragma Export (C, u00058, "system__case_utilB");
u00059 : constant Version_32 := 16#392e2d56#;
pragma Export (C, u00059, "system__case_utilS");
u00060 : constant Version_32 := 16#84a27f0d#;
pragma Export (C, u00060, "interfaces__c_streamsB");
u00061 : constant Version_32 := 16#8bb5f2c0#;
pragma Export (C, u00061, "interfaces__c_streamsS");
u00062 : constant Version_32 := 16#6db6928f#;
pragma Export (C, u00062, "system__crtlS");
u00063 : constant Version_32 := 16#4e6a342b#;
pragma Export (C, u00063, "system__file_ioB");
u00064 : constant Version_32 := 16#ba56a5e4#;
pragma Export (C, u00064, "system__file_ioS");
u00065 : constant Version_32 := 16#b7ab275c#;
pragma Export (C, u00065, "ada__finalizationB");
u00066 : constant Version_32 := 16#19f764ca#;
pragma Export (C, u00066, "ada__finalizationS");
u00067 : constant Version_32 := 16#95817ed8#;
pragma Export (C, u00067, "system__finalization_rootB");
u00068 : constant Version_32 := 16#52d53711#;
pragma Export (C, u00068, "system__finalization_rootS");
u00069 : constant Version_32 := 16#769e25e6#;
pragma Export (C, u00069, "interfaces__cB");
u00070 : constant Version_32 := 16#4a38bedb#;
pragma Export (C, u00070, "interfaces__cS");
u00071 : constant Version_32 := 16#07e6ee66#;
pragma Export (C, u00071, "system__os_libB");
u00072 : constant Version_32 := 16#d7b69782#;
pragma Export (C, u00072, "system__os_libS");
u00073 : constant Version_32 := 16#1a817b8e#;
pragma Export (C, u00073, "system__stringsB");
u00074 : constant Version_32 := 16#639855e7#;
pragma Export (C, u00074, "system__stringsS");
```

```

u00075 : constant Version_32 := 16#e0b8de29#;
pragma Export (C, u00075, "system__file_control_blockS");
u00076 : constant Version_32 := 16#b5b2aca1#;
pragma Export (C, u00076, "system__finalization_mastersB");
u00077 : constant Version_32 := 16#69316dc1#;
pragma Export (C, u00077, "system__finalization_mastersS");
u00078 : constant Version_32 := 16#57a37a42#;
pragma Export (C, u00078, "system__address_imageB");
u00079 : constant Version_32 := 16#bccbd9bb#;
pragma Export (C, u00079, "system__address_imageS");
u00080 : constant Version_32 := 16#7268f812#;
pragma Export (C, u00080, "system__img_boolB");
u00081 : constant Version_32 := 16#e8fe356a#;
pragma Export (C, u00081, "system__img_boolS");
u00082 : constant Version_32 := 16#d7aac20c#;
pragma Export (C, u00082, "system__ioB");
u00083 : constant Version_32 := 16#8365b3ce#;
pragma Export (C, u00083, "system__ioS");
u00084 : constant Version_32 := 16#6d4d969a#;
pragma Export (C, u00084, "system__storage_poolsB");
u00085 : constant Version_32 := 16#e87cc305#;
pragma Export (C, u00085, "system__storage_poolsS");
u00086 : constant Version_32 := 16#e34550ca#;
pragma Export (C, u00086, "system__pool_globalB");
u00087 : constant Version_32 := 16#c88d2d16#;
pragma Export (C, u00087, "system__pool_globalsS");
u00088 : constant Version_32 := 16#9d39c675#;
pragma Export (C, u00088, "system__memoryB");
u00089 : constant Version_32 := 16#445a22b5#;
pragma Export (C, u00089, "system__memoryS");
u00090 : constant Version_32 := 16#6a859064#;
pragma Export (C, u00090, "system__storage_pools__subpoolsB");
u00091 : constant Version_32 := 16#e3b008dc#;
pragma Export (C, u00091, "system__storage_pools__subpoolsS");
u00092 : constant Version_32 := 16#63f11652#;
pragma Export (C, u00092, "system__storage_pools__subpools__finalizationB");
u00093 : constant Version_32 := 16#fe2f4b3a#;
pragma Export (C, u00093, "system__storage_pools__subpools__finalizationS");

-- BEGIN ELABORATION ORDER
-- ada%s
-- interfaces%s
-- system%s
-- system.case_util%s
-- system.case_util%b
-- system.htable%s
-- system.img_bool%s

```

```
-- system.img_bool%b
-- system.img_int%s
-- system.img_int%b
-- system.io%s
-- system.io%b
-- system.parameters%s
-- system.parameters%b
-- system.crtl%s
-- interfaces.c_streams%s
-- interfaces.c_streams%b
-- system.standard_library%s
-- system.exceptions_debug%s
-- system.exceptions_debug%b
-- system.storage_elements%s
-- system.storage_elements%b
-- system.stack_checking%s
-- system.stack_checking%b
-- system.string_hash%s
-- system.string_hash%b
-- system.htable%b
-- system.strings%s
-- system.strings%b
-- system.os_lib%s
-- system.traceback_entries%s
-- system.traceback_entries%b
-- ada.exceptions%s
-- system.soft_links%s
-- system.unsigned_types%s
-- system.val_llu%s
-- system.val_util%s
-- system.val_util%b
-- system.val_llu%b
-- system.wch_con%s
-- system.wch_con%b
-- system.wch_cnv%s
-- system.wch_jis%s
-- system.wch_jis%b
-- system.wch_cnv%b
-- system.wch_stw%s
-- system.wch_stw%b
-- ada.exceptions.last_chance_handler%s
-- ada.exceptions.last_chance_handler%b
-- system.address_image%s
-- system.exception_table%s
-- system.exception_table%b
-- ada.io_exceptions%s
-- ada.tags%s
```

```

-- ada.streams%s
-- ada.streams%b
-- interfaces.c%s
-- system.exceptions%s
-- system.exceptions%b
-- system.exceptions.machine%s
-- system.finalization_root%s
-- system.finalization_root%b
-- ada.finalization%s
-- ada.finalization%b
-- system.storage_pools%s
-- system.storage_pools%b
-- system.finalization_masters%s
-- system.storage_pools.subpools%s
-- system.storage_pools.subpools.finalization%s
-- system.storage_pools.subpools.finalization%b
-- system.memory%s
-- system.memory%b
-- system.standard_library%b
-- system.pool_global%s
-- system.pool_global%b
-- system.file_control_block%s
-- system.file_io%s
-- system.secondary_stack%s
-- system.file_io%b
-- system.storage_pools.subpools%b
-- system.finalization_masters%b
-- interfaces.c%b
-- ada.tags%b
-- system.soft_links%b
-- system.os_lib%b
-- system.secondary_stack%b
-- system.address_image%b
-- system.traceback%s
-- ada.exceptions%b
-- system.traceback%b
-- ada.text_io%s
-- ada.text_io%b
-- hello%b
-- END ELABORATION ORDER

end ada_main;

pragma Ada_95;
-- The following source file name pragmas allow the generated file
-- names to be unique for different main programs. They are needed
-- since the package name will always be Ada_Main.

```

```

pragma Source_File_Name (ada_main, Spec_File_Name => "b~hello.ads");
pragma Source_File_Name (ada_main, Body_File_Name => "b~hello.adb");

pragma Suppress (Overflow_Check);
with Ada.Exceptions;

--  Generated package body for Ada_Main starts here

package body ada_main is
  pragma Warnings (Off);

  --  These values are reference counters associated with units that have
  --  been elaborated. They are used to avoid elaborating the
  --  same unit twice.

  E72 : Short_Integer; pragma Import (Ada, E72, "system__os_lib_E");
  E13 : Short_Integer; pragma Import (Ada, E13, "system__soft_links_E");
  E23 : Short_Integer; pragma Import (Ada, E23, "system__exception_table_E");
  E46 : Short_Integer; pragma Import (Ada, E46, "ada__io_exceptions_E");
  E48 : Short_Integer; pragma Import (Ada, E48, "ada__tags_E");
  E45 : Short_Integer; pragma Import (Ada, E45, "ada__streams_E");
  E70 : Short_Integer; pragma Import (Ada, E70, "interfaces__c_E");
  E25 : Short_Integer; pragma Import (Ada, E25, "system__exceptions_E");
  E68 : Short_Integer; pragma Import (Ada, E68, "system__finalization_root_E");
  E66 : Short_Integer; pragma Import (Ada, E66, "ada__finalization_E");
  E85 : Short_Integer; pragma Import (Ada, E85, "system__storage_pools_E");
  E77 : Short_Integer; pragma Import (Ada, E77, "system__finalization_masters_E");
  E91 : Short_Integer; pragma Import (Ada, E91, "system__storage_pools__subpools_E");
  E87 : Short_Integer; pragma Import (Ada, E87, "system__pool_global_E");
  E75 : Short_Integer; pragma Import (Ada, E75, "system__file_control_block_E");
  E64 : Short_Integer; pragma Import (Ada, E64, "system__file_io_E");
  E17 : Short_Integer; pragma Import (Ada, E17, "system__secondary_stack_E");
  E06 : Short_Integer; pragma Import (Ada, E06, "ada__text_io_E");

  Local_Priority_Specific_Dispatching : constant String := "";
  Local_Interrupt_States : constant String := "";

  Is_Elaborated : Boolean := False;

  procedure finalize_library is
  begin
    E06 := E06 - 1;
    declare
      procedure F1;
      pragma Import (Ada, F1, "ada__text_io__finalize_spec");
    begin

```

```

        F1;
    end;
    E77 := E77 - 1;
    E91 := E91 - 1;
    declare
        procedure F2;
        pragma Import (Ada, F2, "system__file_io__finalize_body");
    begin
        E64 := E64 - 1;
        F2;
    end;
    declare
        procedure F3;
        pragma Import (Ada, F3, "system__file_control_block__finalize_spec");
    begin
        E75 := E75 - 1;
        F3;
    end;
    E87 := E87 - 1;
    declare
        procedure F4;
        pragma Import (Ada, F4, "system__pool_global__finalize_spec");
    begin
        F4;
    end;
    declare
        procedure F5;
        pragma Import (Ada, F5, "system__storage_pools__subpools__finalize_spec");
    begin
        F5;
    end;
    declare
        procedure F6;
        pragma Import (Ada, F6, "system__finalization_masters__finalize_spec");
    begin
        F6;
    end;
    declare
        procedure Reraise_Library_Exception_If_Any;
        pragma Import (Ada, Reraise_Library_Exception_If_Any, "__gnat_reraise_library_exception_if_any");
    begin
        Reraise_Library_Exception_If_Any;
    end;
end finalize_library;

-----
-- adainit --

```

-----

procedure adainit is

```

    Main_Priority : Integer;
    pragma Import (C, Main_Priority, "__gl_main_priority");
    Time_Slice_Value : Integer;
    pragma Import (C, Time_Slice_Value, "__gl_time_slice_val");
    WC_Encoding : Character;
    pragma Import (C, WC_Encoding, "__gl_wc_encoding");
    Locking_Policy : Character;
    pragma Import (C, Locking_Policy, "__gl_locking_policy");
    Queuing_Policy : Character;
    pragma Import (C, Queuing_Policy, "__gl_queuing_policy");
    Task_Dispatching_Policy : Character;
    pragma Import (C, Task_Dispatching_Policy, "__gl_task_dispatching_policy");
    Priority_Specific_Dispatching : System.Address;
    pragma Import (C, Priority_Specific_Dispatching, "__gl_priority_specific_dispatcher");
    Num_Specific_Dispatching : Integer;
    pragma Import (C, Num_Specific_Dispatching, "__gl_num_specific_dispatching");
    Main_CPU : Integer;
    pragma Import (C, Main_CPU, "__gl_main_cpu");
    Interrupt_States : System.Address;
    pragma Import (C, Interrupt_States, "__gl_interrupt_states");
    Num_Interrupt_States : Integer;
    pragma Import (C, Num_Interrupt_States, "__gl_num_interrupt_states");
    Unreserve_All_Interrupts : Integer;
    pragma Import (C, Unreserve_All_Interrupts, "__gl_unreserve_all_interrupts");
    Detect_Blocking : Integer;
    pragma Import (C, Detect_Blocking, "__gl_detect_blocking");
    Default_Stack_Size : Integer;
    pragma Import (C, Default_Stack_Size, "__gl_default_stack_size");
    Leap_Seconds_Support : Integer;
    pragma Import (C, Leap_Seconds_Support, "__gl_leap_seconds_support");

    procedure Runtime_Initialize;
    pragma Import (C, Runtime_Initialize, "__gnat_runtime_initialize");

    Finalize_Library_Objects : No_Param_Proc;
    pragma Import (C, Finalize_Library_Objects, "__gnat_finalize_library_objects");

```

-- Start of processing for adainit

begin

```

    -- Record various information for this partition. The values
    -- are derived by the binder from information stored in the ali

```

```

-- files by the compiler.

if Is_Elaborated then
    return;
end if;
Is_Elaborated := True;
Main_Priority := -1;
Time_Slice_Value := -1;
WC_Encoding := 'b';
Locking_Policy := ' ';
Queuing_Policy := ' ';
Task_Dispatching_Policy := ' ';
Priority_Specific_Dispatching :=
    Local_Priority_Specific_Dispatching'Address;
Num_Specific_Dispatching := 0;
Main_CPU := -1;
Interrupt_States := Local_Interrupt_States'Address;
Num_Interrupt_States := 0;
Unreserve_All_Interrupts := 0;
Detect_Blocking := 0;
Default_Stack_Size := -1;
Leap_Seconds_Support := 0;

Runtime_Initialize;

Finalize_Library_Objects := finalize_library'access;

-- Now we have the elaboration calls for all units in the partition.
-- The Elab_Spec and Elab_Body attributes generate references to the
-- implicit elaboration procedures generated by the compiler for
-- each unit that requires elaboration. Also increment a reference
-- counter for each unit.

System.Soft_Links'Elab_Spec;
System.Exception_Table'Elab_Body;
E23 := E23 + 1;
Ada.Io_Exceptions'Elab_Spec;
E46 := E46 + 1;
Ada.Tags'Elab_Spec;
Ada.Streams'Elab_Spec;
E45 := E45 + 1;
Interfaces.C'Elab_Spec;
System.Exceptions'Elab_Spec;
E25 := E25 + 1;
System.Finalization_Root'Elab_Spec;
E68 := E68 + 1;
Ada.Finalization'Elab_Spec;

```

```

E66 := E66 + 1;
System.Storage_Pools'Elab_Spec;
E85 := E85 + 1;
System.Finalization_Masters'Elab_Spec;
System.Storage_Pools.Subpools'Elab_Spec;
System.Pool_Global'Elab_Spec;
E87 := E87 + 1;
System.File_Control_Block'Elab_Spec;
E75 := E75 + 1;
System.File_Io'Elab_Body;
E64 := E64 + 1;
E91 := E91 + 1;
System.Finalization_Masters'Elab_Body;
E77 := E77 + 1;
E70 := E70 + 1;
Ada.Tags'Elab_Body;
E48 := E48 + 1;
System.Soft_Links'Elab_Body;
E13 := E13 + 1;
System.Os_Lib'Elab_Body;
E72 := E72 + 1;
System.Secondary_Stack'Elab_Body;
E17 := E17 + 1;
Ada.Text_Io'Elab_Spec;
Ada.Text_Io'Elab_Body;
E06 := E06 + 1;
end adainit;

-----
-- adafinal --
-----

procedure adafinal is
  procedure s_stalib_adafinal;
  pragma Import (C, s_stalib_adafinal, "system__standard_library__adafinal");

  procedure Runtime_Finalize;
  pragma Import (C, Runtime_Finalize, "__gnat_runtime_finalize");

begin
  if not Is_Elaborated then
    return;
  end if;
  Is_Elaborated := False;
  Runtime_Finalize;
  s_stalib_adafinal;
end adafinal;

```

```

-- We get to the main program of the partition by using
-- pragma Import because if we try to 'with' the unit and
-- call it in Ada style, not only do we waste time recompiling it,
-- but we don't know the right switches (e.g.@: identifier
-- character set) to be used to compile it.

procedure Ada_Main_Program;
pragma Import (Ada, Ada_Main_Program, "_ada_hello");

-----
-- main --
-----

-- main is actually a function, as in the ANSI C standard,
-- defined to return the exit status. The three parameters
-- are the argument count, argument values and environment
-- pointer.

function main
  (argc : Integer;
   argv : System.Address;
   envp : System.Address)
  return Integer
is
  -- The initialize routine performs low level system
  -- initialization using a standard library routine which
  -- sets up signal handling and performs any other
  -- required setup. The routine can be found in file
  -- a-init.c.

  procedure initialize;
  pragma Import (C, initialize, "__gnat_initialize");

  -- The finalize routine performs low level system
  -- finalization using a standard library routine. The
  -- routine is found in file a-final.c and in the standard
  -- distribution is a dummy routine that does nothing, so
  -- really this is a hook for special user finalization.

  procedure finalize;
  pragma Import (C, finalize, "__gnat_finalize");

  -- The following is to initialize the SEH exceptions

  SEH : aliased array (1 .. 2) of Integer;

```

```

    Ensure_Reference : aliased System.Address := Ada_Main_Program_Name'Address;
    pragma Volatile (Ensure_Reference);

-- Start of processing for main

begin
    -- Save global variables

    gnat_argc := argc;
    gnat_argv := argv;
    gnat_envp := envp;

    -- Call low level system initialization

    Initialize (SEH'Address);

    -- Call our generated Ada initialization routine

    adainit;

    -- Now we call the main program of the partition

    Ada_Main_Program;

    -- Perform Ada finalization

    adafinal;

    -- Perform low level system finalization

    Finalize;

    -- Return the proper exit status
    return (gnat_exit_status);
end;

-- This section is entirely comments, so it has no effect on the
-- compilation of the Ada_Main package. It provides the list of
-- object files and linker options, as well as some standard
-- libraries needed for the link. The gnatlink utility parses
-- this b~hello.adb file to read these comment lines to generate
-- the appropriate command line arguments for the call to the
-- system linker. The BEGIN/END lines are used for sentinels for
-- this parsing operation.

-- The exact file names will of course depend on the environment,
-- host/target and location of files on the host system.

```

```
-- BEGIN Object file/option list
--      ./hello.o
--      -L./
--      -L/usr/local/gnat/lib/gcc-lib/i686-pc-linux-gnu/2.8.1/adalib/
--      /usr/local/gnat/lib/gcc-lib/i686-pc-linux-gnu/2.8.1/adalib/libgnat.a
-- END Object file/option list

      end ada_main;
```

The Ada code in the above example is exactly what is generated by the binder. We have added comments to more clearly indicate the function of each part of the generated `Ada_Main` package.

The code is standard Ada in all respects, and can be processed by any tools that handle Ada. In particular, you can use the debugger in Ada mode to debug the generated `Ada_Main` package. For example, suppose that for reasons you don't understand, your program is crashing during elaboration of the body of `Ada.Text_IO`. To locate this bug, you can place a breakpoint on the call:

```
      Ada.Text_IO'Elab_Body;
```

and trace the elaboration routine for this package to find out where the problem might be (more usually, of course, you would be debugging elaboration code in your own application).

## 9 Elaboration Order Handling in GNAT

This appendix describes the handling of elaboration code in Ada and GNAT and discusses how you can control the order of elaboration of program units in GNAT, either automatically or with explicit programming features.

### 9.1 Elaboration Code

Ada defines the term ‘execution’ as the process by which a construct achieves its run-time effect. This process is also referred to as ‘elaboration’ for declarations and ‘evaluation’ for expressions.

The execution model in Ada allows for certain sections of an Ada program to be executed prior to execution of the program itself, primarily with the intent of initializing data. These sections are referred to as ‘elaboration code’. Elaboration code is executed as follows:

- \* All partitions of an Ada program are executed in parallel with one another, possibly in a separate address space and possibly on a separate computer.
- \* The execution of a partition involves running the environment task for that partition.
- \* The environment task executes all elaboration code (if available) for all units within that partition. This code is said to be executed at ‘elaboration time’.
- \* The environment task executes the Ada program (if available) for that partition.

In addition to the Ada terminology, this appendix defines the following terms:

- \* ‘Invocation’

The act of calling a subprogram, instantiating a generic, or activating a task.

- \* ‘Scenario’

A construct that is elaborated or invoked by elaboration code is referred to as an ‘elaboration scenario’ or simply a ‘scenario’. GNAT recognizes the following scenarios:

- ‘Access’ of entries, operators, and subprograms
- Activation of tasks
- Calls to entries, operators, and subprograms
- Instantiations of generic templates

- \* ‘Target’

A construct elaborated by a scenario is referred to as an ‘elaboration target’ or simply a ‘target’. GNAT recognizes the following targets:

- For ‘Access’ of entries, operators, and subprograms, the target is the entry, operator, or subprogram being aliased.
- For activation of tasks, the target is the task body
- For calls to entries, operators, and subprograms, the target is the entry, operator, or subprogram being invoked.
- For instantiations of generic templates, the target is the generic template being instantiated.

Elaboration code may appear in two distinct contexts:

\* ‘Library level’

A scenario appears at the library level when it is encapsulated by a package [body] compilation unit, ignoring any other package [body] declarations in between.

```
with Server;
package Client is
  procedure Proc;

  package Nested is
    Val : ... := Server.Func;
  end Nested;
end Client;
```

In the example above, the call to `Server.Func` is an elaboration scenario because it appears at the library level of package `Client`. Note that the declaration of package `Nested` is ignored according to the definition given above. As a result, the call to `Server.Func` will be invoked when the spec of unit `Client` is elaborated.

\* ‘Package body statements’

A scenario appears within the statement sequence of a package body when it is bounded by the region starting from the `begin` keyword of the package body and ending at the `end` keyword of the package body.

```
package body Client is
  procedure Proc is
  begin
    ...
  end Proc;
begin
  Proc;
end Client;
```

In the example above, the call to `Proc` is an elaboration scenario because it appears within the statement sequence of package body `Client`. As a result, the call to `Proc` will be invoked when the body of `Client` is elaborated.

## 9.2 Elaboration Order

The sequence by which the elaboration code of all units within a partition is executed is referred to as ‘elaboration order’.

Within a single unit, elaboration code is executed in sequential order.

```
package body Client is
  Result : ... := Server.Func;

  procedure Proc is
    package Inst is new Server.Gen;
  begin
    Inst.Eval (Result);
  end Proc;
```

```

begin
  Proc;
end Client;

```

In the example above, the elaboration order within package body `Client` is as follows:

1. The object declaration of `Result` is elaborated.
  - \* Function `Server.Func` is invoked.
2. The subprogram body of `Proc` is elaborated.
3. Procedure `Proc` is invoked.
  - \* Generic unit `Server.Gen` is instantiated as `Inst`.
  - \* Instance `Inst` is elaborated.
  - \* Procedure `Inst.Eval` is invoked.

The elaboration order of all units within a partition depends on the following factors:

- \* ‘with’ed units
- \* parent units
- \* purity of units
- \* preelaborability of units
- \* presence of elaboration-control pragmas
- \* invocations performed in elaboration code

A program may have several possible elaboration orders depending on its structure:

```

package Server is
  function Func (Index : Integer) return Integer;
end Server;

package body Server is
  Results : array (1 .. 5) of Integer := (1, 2, 3, 4, 5);

  function Func (Index : Integer) return Integer is
  begin
    return Results (Index);
  end Func;
end Server;

with Server;
package Client is
  Val : constant Integer := Server.Func (3);
end Client;

with Client;
procedure Main is begin null; end Main;

```

The following elaboration order exhibits a fundamental problem referred to as ‘access-before-elaboration’ or simply ‘ABE’.

```

spec of Server
spec of Client
body of Server

```

body of Main

The elaboration of `Server`'s spec materializes function `Func`, making it callable. The elaboration of `Client`'s spec elaborates the declaration of `Val`. This invokes function `Server.Func`, however the body of `Server.Func` has not been elaborated yet because `Server`'s body comes after `Client`'s spec in the elaboration order. As a result, the value of constant `Val` is now undefined.

Without any guarantees from the language, an undetected ABE problem may hinder proper initialization of data, which in turn may lead to undefined behavior at run time. To prevent such ABE problems, Ada employs dynamic checks in the same vein as index or null exclusion checks. A failed ABE check raises exception `Program_Error`.

The following elaboration order avoids the ABE problem and the program can be successfully elaborated.

spec of Server  
body of Server  
spec of Client  
body of Main

Ada states that a total elaboration order must exist, but it does not define what this order is. A compiler is thus tasked with choosing a suitable elaboration order which satisfies the dependencies imposed by 'with' clauses, unit categorization, elaboration-control pragmas, and invocations performed in elaboration code. Ideally, an order that avoids ABE problems should be chosen, however a compiler may not always find such an order due to complications with respect to control and data flow.

### 9.3 Checking the Elaboration Order

To avoid placing the entire elaboration-order burden on the programmer, Ada provides three lines of defense:

- \* 'Static semantics'

Static semantic rules restrict the possible choice of elaboration order. For instance, if unit `Client` 'with's unit `Server`, then the spec of `Server` is always elaborated prior to `Client`. The same principle applies to child units - the spec of a parent unit is always elaborated prior to the child unit.

- \* 'Dynamic semantics'

Dynamic checks are performed at run time to ensure that a target is elaborated prior to a scenario that invokes it, thus avoiding ABE problems. A failed run-time check raises exception `Program_Error`. The following restrictions apply:

- 'Restrictions on calls'

An entry, operator, or subprogram can be called from elaboration code only when the corresponding body has been elaborated.

- 'Restrictions on instantiations'

A generic unit can be instantiated by elaboration code only when the corresponding body has been elaborated.

- 'Restrictions on task activation'

A task can be activated by elaboration code only when the body of the associated task type has been elaborated.

The restrictions above can be summarized by the following rule:

‘If a target has a body, then this body must be elaborated prior to the scenario that invokes the target.’

- \* ‘Elaboration control’

Ada provides pragmas for you to specify the desired elaboration order.

## 9.4 Controlling the Elaboration Order in Ada

Ada provides several idioms and pragmas to aid you in specifying your desired elaboration order and avoiding ABE problems.

- \* ‘Packages without a body’

A library package that does not require a completing body does not suffer from ABE problems.

```
package Pack is
  generic
    type Element is private;
  package Containers is
    type Element_Array is array (1 .. 10) of Element;
  end Containers;
end Pack;
```

In the example above, package `Pack` does not require a body because it does not contain any constructs which require completion in a body. As a result, generic `Pack.Containers` can be instantiated without encountering any ABE problems.

- \* ‘pragma `Pure`’

Pragma `Pure` places sufficient restrictions on a unit to guarantee that no scenario within the unit can result in an ABE problem.

- \* ‘pragma `Preelaborate`’

Pragma `Preelaborate` is slightly less restrictive than pragma `Pure`, but still strong enough to prevent ABE problems within a unit.

- \* ‘pragma `Elaborate_Body`’

Pragma `Elaborate_Body` requires that the body of a unit is elaborated immediately after its spec. This restriction guarantees that no client scenario can invoke a server target before the target body has been elaborated because the spec and body are effectively “glued” together.

```
package Server is
  pragma Elaborate_Body;

  function Func return Integer;
end Server;

package body Server is
  function Func return Integer is
  begin
    ...
  end Func;
```

```

end Server;
with Server;
package Client is
  Val : constant Integer := Server.Func;
end Client;

```

In the example above, pragma `Elaborate_Body` guarantees the following elaboration order:

```

spec of Server
body of Server
spec of Client

```

because the spec of `Server` must be elaborated prior to `Client` by virtue of the ‘with’ clause and the body of `Server` must be elaborated immediately after the spec of `Server`.

Removing pragma `Elaborate_Body` could result in the following incorrect elaboration order:

```

spec of Server
spec of Client
body of Server

```

where `Client` invokes `Server.Func`, but the body of `Server.Func` has not been elaborated yet.

The pragmas outlined above allow a server unit to guarantee safe elaboration use by client units. Thus it is a good rule to mark units as `Pure` or `Preelaborate`, and if this is not possible, mark them as `Elaborate_Body`.

There are however situations where `Pure`, `Preelaborate`, and `Elaborate_Body` are not applicable. Ada provides another set of pragmas for use by client units to help ensure the elaboration safety of server units they depend on.

\* ‘pragma `Elaborate (Unit)`’

You can place pragma `Elaborate` in the context clauses of a unit, after a ‘with’ clause. It guarantees that both the spec and body of its argument will be elaborated prior to the unit with the pragma. Note that other unrelated units may be elaborated in between the spec and the body.

```

package Server is
  function Func return Integer;
end Server;

package body Server is
  function Func return Integer is
  begin
    ...
  end Func;
end Server;

with Server;
pragma Elaborate (Server);
package Client is
  Val : constant Integer := Server.Func;
end Client;

```

In the example above, pragma `Elaborate` guarantees the following elaboration order:

```
spec of Server
body of Server
spec of Client
```

Removing pragma `Elaborate` could result in the following incorrect elaboration order:

```
spec of Server
spec of Client
body of Server
```

where `Client` invokes `Server.Func`, but the body of `Server.Func` has not been elaborated yet.

\* ‘pragma `Elaborate_All (Unit)`’

You can place pragma `Elaborate_All` in the context clauses of a unit, after a ‘with’ clause. It guarantees that both the spec and body of its argument will be elaborated prior to the unit with the pragma as well as all units ‘with’ed by the spec and body of the argument, recursively. Note that other unrelated units may be elaborated in between the spec and the body.

```
package Math is
  function Factorial (Val : Natural) return Natural;
end Math;

package body Math is
  function Factorial (Val : Natural) return Natural is
  begin
    ...;
  end Factorial;
end Math;

package Computer is
  type Operation_Kind is (None, Op_Factorial);

  function Compute
    (Val : Natural;
     Op  : Operation_Kind) return Natural;
end Computer;

with Math;
package body Computer is
  function Compute
    (Val : Natural;
     Op  : Operation_Kind) return Natural
  is
    if Op = Op_Factorial then
      return Math.Factorial (Val);
    end if;

    return 0;
  end Compute;
```

```

end Computer;
with Computer;
pragma Elaborate_All (Computer);
package Client is
  Val : constant Natural :=
    Computer.Compute (123, Computer.Op_Factorial);
end Client;

```

In the example above, `pragma Elaborate_All` can result in the following elaboration order:

```

spec of Math
body of Math
spec of Computer
body of Computer
spec of Client

```

Note that there are several allowable suborders for the specs and bodies of `Math` and `Computer`, but the point is that these specs and bodies will be elaborated prior to `Client`.

Removing `pragma Elaborate_All` could result in the following incorrect elaboration order:

```

spec of Math
spec of Computer
body of Computer
spec of Client
body of Math

```

where `Client` invokes `Computer.Compute`, which in turn invokes `Math.Factorial`, but the body of `Math.Factorial` has not been elaborated yet.

All pragmas shown above can be summarized by the following rule:

‘If a client unit elaborates a server target directly or indirectly, then if the server unit requires a body and does not have `pragma Pure`, `Preelaborate`, or `Elaborate_Body`, then the client unit should have `pragma Elaborate` or `Elaborate_All` for the server unit.’

If you do not follow the rule outlined above, a program may fall in one of the following ways:

- \* ‘No elaboration order exists’

In this case a compiler must diagnose the situation and refuse to build an executable program.

- \* ‘One or more incorrect elaboration orders exist’

In this case a compiler can build an executable program, but `Program_Error` will be raised when the program is run.

- \* ‘Several elaboration orders exist, some correct, some incorrect’

In this case, you have not controlled the elaboration order. As a result, a compiler may or may not pick one of the correct orders and the program may or may not raise `Program_Error` when it is run. This is the worst possible state because the program may fail on another compiler or even a different version of the same compiler.

\* ‘One or more correct orders exist’

In this case a compiler can build an executable program and the program is run successfully. This state may be guaranteed by following the outlined rules or may be the result of good program architecture.

Note that one additional advantage of using **Elaborate** and **Elaborate\_All** is that the program continues to stay in the last state (one or more correct orders exist) even if maintenance changes the bodies of targets.

## 9.5 Controlling the Elaboration Order in GNAT

In addition to Ada semantics and rules synthesized from them, GNAT offers three elaboration models to aid you in specifying the correct elaboration order and in diagnosing elaboration problems.

\* ‘Dynamic elaboration model’

This is the most permissive of the three elaboration models and emulates the behavior specified by the Ada Reference Manual. When the dynamic model is in effect, GNAT makes the following assumptions:

- All code within all units in a partition is considered to be elaboration code.
- Some of the invocations in elaboration code may not take place at run time due to conditional execution.

GNAT performs extensive diagnostics on a unit-by-unit basis for all scenarios that invoke internal targets. In addition, GNAT generates run-time checks for all external targets and for all scenarios that may exhibit ABE problems.

The elaboration order is obtained by honoring all ‘with’ clauses, purity and preelaborability of units, and elaboration-control pragmas. The dynamic model attempts to take all invocations in elaboration code into account. If an invocation leads to a circularity, GNAT ignores the invocation based on the assumptions stated above. An order obtained using the dynamic model may fail an ABE check at run time when GNAT ignored an invocation.

You enable the dynamic model with the compiler switch **-gnatE**.

\* ‘Static elaboration model’

This is the middle ground of the three models. When the static model is in effect, GNAT makes the following assumptions:

- Only code at the library level and in package body statements within all units in a partition is considered to be elaboration code.
- All invocations in elaboration will take place at run time, regardless of conditional execution.

GNAT performs extensive diagnostics on a unit-by-unit basis for all scenarios that invoke internal targets. In addition, GNAT generates run-time checks for all external targets and for all scenarios that may exhibit ABE problems.

The elaboration order is obtained by honoring all ‘with’ clauses, purity and preelaborability of units, presence of elaboration-control pragmas, and all invocations in elaboration code. An order obtained using the static model is guaranteed to be ABE problem-free, excluding dispatching calls and access-to-subprogram types.

The static model is the default model in GNAT.

\* ‘SPARK elaboration model’

This is the most conservative of the three models and enforces the SPARK rules of elaboration as defined in the SPARK Reference Manual, section 7.7. The SPARK model is in effect only when a scenario and a target reside in a region subject to `SPARK_Mode On`, otherwise the dynamic or static model is in effect.

The SPARK model is enabled with compiler switch `-gnatd.v`.

\* ‘Legacy elaboration models’

In addition to the three elaboration models outlined above, GNAT provides the following legacy models:

- *Legacy elaboration-checking model* available in pre-18.x versions of GNAT. You can enable this model with compiler switch `-gnatH`.
- *Legacy elaboration-order model* available in pre-20.x versions of GNAT. You can enable this model with binder switch `-H`.

You can relax the dynamic, legacy, and static models by specifying compiler switch `-gnatJ`, which makes them more permissive. Note that in this mode, GNAT may not diagnose certain elaboration issues or install run-time checks.

## 9.6 Mixing Elaboration Models

You can mix units compiled with different elaboration models. However you must observe the following rules:

- \* A client unit compiled with the dynamic model can only ‘with’ a server unit that meets at least one of the following criteria:
  - The server unit is compiled with the dynamic model.
  - The server unit is a GNAT implementation unit from the `Ada`, `GNAT`, `Interfaces`, or `System` hierarchies.
  - The server unit has pragma `Pure` or `Preelaborate`.
  - The client unit has an explicit `Elaborate_All` pragma for the server unit.

These rules ensure that elaboration checks are not omitted. If the rules are violated, the binder emits a warning:

```
warning: "x.ads" has dynamic elaboration checks and with's
warning:  "y.ads" which has static elaboration checks
```

You can suppress these warnings by specifying binder switch `-ws`.

## 9.7 ABE Diagnostics

GNAT performs extensive diagnostics on a unit-by-unit basis for all scenarios that invoke internal targets, regardless of whether the dynamic, SPARK, or static model is in effect.

Note that GNAT emits warnings rather than errors whenever it encounters an elaboration problem. This is because the elaboration model in effect may be too conservative or a particular scenario may not be invoked due to conditional execution. You can selectively suppress the warnings with pragma `Warnings (Off)` or globally with compiler switch `-gnatwL`.

A ‘guaranteed ABE’ arises when the body of a target is not elaborated early enough and causes ‘all’ scenarios that directly invoke the target to fail.

```
package body Guaranteed_ABE is
  function ABE return Integer;

  Val : constant Integer := ABE;

  function ABE return Integer is
  begin
    ...
  end ABE;
end Guaranteed_ABE;
```

In the example above, the elaboration of `Guaranteed_ABE`’s body elaborates the declaration of `Val`. This invokes function `ABE`, however the body of `ABE` has not been elaborated yet. GNAT emits the following diagnostic:

```
4.    Val : constant Integer := ABE;
      |
      >>> warning: cannot call "ABE" before body seen
      >>> warning: Program_Error will be raised at run time
```

A ‘conditional ABE’ arises when the body of a target is not elaborated early enough and causes ‘some’ scenarios that directly invoke the target to fail.

```
1. package body Conditional_ABE is
2.   procedure Force_Body is null;
3.
4.   generic
5.     with function Func return Integer;
6.   package Gen is
7.     Val : constant Integer := Func;
8.   end Gen;
9.
10.  function ABE return Integer;
11.
12.  function Cause_ABE return Boolean is
13.    package Inst is new Gen (ABE);
14.  begin
15.    ...
16.  end Cause_ABE;
17.
18.  Val : constant Boolean := Cause_ABE;
19.
20.  function ABE return Integer is
21.  begin
22.    ...
23.  end ABE;
24.
25.  Safe : constant Boolean := Cause_ABE;
```

```
26. end Conditional_ABE;
```

In the example above, the elaboration of package body `Conditional_ABE` elaborates the declaration of `Val`. This invokes function `Cause_ABE`, which instantiates generic unit `Gen` as `Inst`. The elaboration of `Inst` invokes function `ABE`, however the body of `ABE` has not been elaborated yet. GNAT emits the following diagnostic:

```
13.      package Inst is new Gen (ABE);
      |
      >>> warning: in instantiation at line 7
      >>> warning: cannot call "ABE" before body seen
      >>> warning: Program_Error may be raised at run time
      >>> warning:  body of unit "Conditional_ABE" elaborated
      >>> warning:  function "Cause_ABE" called at line 18
      >>> warning:  function "ABE" called at line 7, instance at line 13
```

Note that the same `ABE` problem does not occur with the elaboration of declaration `Safe` because the body of function `ABE` has already been elaborated at that point.

## 9.8 SPARK Diagnostics

GNAT enforces the SPARK rules of elaboration as defined in the SPARK Reference Manual section 7.7 when you specify compiler switch `-gnatd.v`. Note that GNAT emits errors whenever it encounters a violation of the SPARK rules.

```
1. with Server;
2. package body SPARK_Diagnostics with SPARK_Mode is
3.   Val : constant Integer := Server.Func;
      |
      >>> call to "Func" during elaboration in SPARK
      >>> unit "SPARK_Diagnostics" requires pragma "Elaborate_All" for "Server"
      >>>  body of unit "SPARK_Model" elaborated
      >>>  function "Func" called at line 3

4. end SPARK_Diagnostics;
```

## 9.9 Elaboration Circularities

An ‘elaboration circularity’ occurs whenever the elaboration of a set of units enters a dead-locked state, where each unit is waiting for another unit to be elaborated. This situation may be the result of improper use of ‘with’ clauses, elaboration-control pragmas, or invocations in elaboration code.

The following example exhibits an elaboration circularity.

```
with B; pragma Elaborate (B);
package A is
end A;

package B is
  procedure Force_Body;
end B;

with C;
```

```

package body B is
  procedure Force_Body is null;

  Elab : constant Integer := C.Func;
end B;

package C is
  function Func return Integer;
end C;

with A;
package body C is
  function Func return Integer is
  begin
    ...
  end Func;
end C;

```

The binder emits the following diagnostic:

```

error: Elaboration circularity detected
info:
info:   Reason:
info:
info:     unit "a (spec)" depends on its own elaboration
info:
info:   Circularity:
info:
info:     unit "a (spec)" has with clause and pragma Elaborate for unit "b
info:     unit "b (body)" is in the closure of pragma Elaborate
info:     unit "b (body)" invokes a construct of unit "c (body)" at elabora
info:     unit "c (body)" has with clause for unit "a (spec)"
info:
info:   Suggestions:
info:
info:     remove pragma Elaborate for unit "b (body)" in unit "a (spec)"
info:     use the dynamic elaboration model (compiler switch -gnatE)

```

The diagnostic consist of the following sections:

- \* Reason

This section provides a short explanation describing why the set of units could not be ordered.

- \* Circularity

This section enumerates the units comprising the deadlocked set, along with their interdependencies.

- \* Suggestions

This section enumerates various tactics for eliminating the circularity.

## 9.10 Resolving Elaboration Circularities

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 invoked 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.

When faced with an elaboration circularity, you should also consider the tactics given in the suggestions section of the circularity diagnostic. Depending on the units involved in the circularity, their ‘with’ clauses, purity, preelaborability, presence of elaboration-control pragmas and invocations at elaboration time, the binder may suggest one or more of the following tactics to eliminate the circularity:

\* Pragma Elaborate elimination

```
remove pragma Elaborate for unit "..." in unit "..."
```

The binder suggests this tactic when it has determined that:

- pragma `Elaborate` prevents a set of units from being elaborated.
- The removal of the pragma will not eliminate the semantic effects of the pragma. In other words, the argument of the pragma will still be elaborated prior to the unit containing the pragma.
- The removal of the pragma will enable the successful ordering of the units.

You should remove the pragma as advised and rebuild the program.

\* Pragma Elaborate\_All elimination

```
remove pragma Elaborate_All for unit "..." in unit "..."
```

The binder suggests this tactic when it has determined that:

- pragma `Elaborate_All` prevents a set of units from being elaborated.
- The removal of the pragma will not eliminate the semantic effects of the pragma. In other words, the argument of the pragma along with its ‘with’ closure will still be elaborated prior to the unit containing the pragma.
- The removal of the pragma will enable the successful ordering of the units.

You should remove the pragma as advised and rebuild the program.

\* Pragma Elaborate\_All downgrade

```
change pragma Elaborate_All for unit "..." to Elaborate in unit "..."
```

The binder always suggests this tactic when it suggests the pragma `Elaborate_All` elimination tactic. It offers a different alternative of guaranteeing that the argument of the pragma will still be elaborated prior to the unit containing the pragma.

You should update the pragma as advised and rebuild the program.

\* Pragma Elaborate\_Body elimination

```
remove pragma Elaborate_Body in unit "..."
```

The binder suggests this tactic when it has determined that:

- pragma `Elaborate_Body` prevents a set of units from being elaborated.
- The removal of the pragma will enable the successful ordering of the units.

Note that the binder cannot determine whether the pragma is required for other purposes, such as guaranteeing the initialization of a variable declared in the spec by elaboration code in the body.

If the pragma is not required for another purpose, you should remove the pragma as advised and rebuild the program.

\* Use of pragma Restrictions

```
use pragma Restrictions (No_Entry_Calls_In_Elaboration_Code)
```

The binder suggests this tactic when it has determined that a task activation at elaboration time:

- Prevents a set of units from being elaborated.

Note that the binder cannot determine with certainty whether the task will block at elaboration time.

The programmer should create a configuration file, place the pragma within, update the general compilation arguments, and rebuild the program.

\* Use of dynamic elaboration model

```
use the dynamic elaboration model (compiler switch -gnatE)
```

This tactic is suggested when the binder has determined that an invocation at elaboration time:

- Prevents a set of units from being elaborated.
- The use of the dynamic model will enable the successful ordering of the units.

You have two options:

- Determine the units involved in the invocation using the detailed invocation information and add compiler switch `-gnatE` to the compilation arguments of those units only. This approach will yield safer elaboration orders compared to the other option because it will minimize the opportunities presented to the dynamic model for ignoring invocations.
- Add compiler switch `-gnatE` to the global compilation arguments.

\* Use of detailed invocation information

```
use detailed invocation information (compiler switch -gnatd_F)
```

The binder always suggests this tactic when it suggests use of the dynamic model tactic. It causes the circularity section of the circularity diagnostic to describe the flow of elaboration code from a unit to a unit, enumerating all such paths in the process.

You should analyze this information to determine which units should be compiled with the dynamic model.

\* Forced-dependency elimination

```
remove the dependency of unit "..." on unit "..." from the argument of switch -f
```

The binder suggests this tactic when it has determined that a dependency present in the forced-elaboration-order file indicated by binder switch `-f`:

- Prevents a set of units from being elaborated.
- The removal of the dependency will enable the successful ordering of the units.

You should edit the forced-elaboration-order file, remove the dependency, and rebind the program.

- \* All forced-dependency elimination

`remove switch -f`

The binder suggests this tactic when editing the forced-elaboration-order file is not an option.

You should remove binder switch `-f` from the binder arguments and rebind.

- \* Multiple-circularities diagnostic

`diagnose all circularities (binder switch -d_C)`

By default, the binder only diagnoses the highest-precedence circularity. If the program contains multiple circularities, the binder will suggest the use of binder switch `-d_C` in order to obtain the diagnostics for each circularity.

You should add binder switch `-d_C` to the binder arguments and rebind.

If none of the tactics suggested by the binder eliminate the elaboration circularity, you should consider using one of the legacy elaboration models, in the following order:

- \* Use the pre-20.x legacy elaboration-order model, with binder switch `-H`.
- \* Use both pre-18.x and pre-20.x legacy elaboration models, with compiler switch `-gnatH` and binder switch `-H`.
- \* Use the relaxed static-elaboration model, with compiler switches `-gnatH -gnatJ` and binder switch `-H`.
- \* Use the relaxed dynamic-elaboration model, with compiler switches `-gnatH -gnatJ -gnatE` and binder switch `-H`.

## 9.11 Elaboration-related Compiler Switches

GNAT has several switches that affect the elaboration model and consequently the elaboration order chosen by the binder.

### `-gnatE`

Dynamic elaboration checking mode enabled

When this switch is in effect, GNAT activates the dynamic model.

### `-gnatEl`

Turn on informational messages on generated `Elaborate[_All]` pragmas

This switch is only applicable to the pre-20.x legacy elaboration models. The post-20.x elaboration model no longer relies on implicitly generated `Elaborate` and `Elaborate_All` pragmas to order units.

When this switch is in effect, GNAT will emit the following supplementary information depending on the elaboration model in effect.

- ‘Dynamic model’

GNAT will indicate missing `Elaborate` and `Elaborate_All` pragmas for all library-level scenarios within the partition.

- ‘Static model’

GNAT will indicate all scenarios invoked during elaboration. In addition, it will provide detailed traceback when an implicit `Elaborate` or `Elaborate_All` pragma is generated.

- ‘SPARK model’

GNAT will indicate how an elaboration requirement is met by the context of a unit. This diagnostic requires compiler switch `-gnatd.v`.

```

1. with Server; pragma Elaborate_All (Server);
2. package Client with SPARK_Mode is
3.   Val : constant Integer := Server.Func;
      |
   >>> info: call to "Func" during elaboration in SPARK
   >>> info: "Elaborate_All" requirement for unit "Server" met by prag
4. end Client;
```

`-gnatH`

Legacy elaboration checking mode enabled

When this switch is in effect, GNAT will utilize the pre-18.x elaboration model.

`-gnatJ`

Relaxed elaboration checking mode enabled

When this switch is in effect, GNAT will not process certain scenarios, resulting in a more permissive elaboration model. Note that this may eliminate some diagnostics and run-time checks.

`-gnatw.f`

Turn on warnings for suspicious `Subp'Access`

When this switch is in effect, GNAT will treat `'Access` of an entry, operator, or subprogram as a potential call to the target and issue warnings:

```

1. package body Attribute_Call is
2.   function Func return Integer;
3.   type Func_Ptr is access function return Integer;
4.
5.   Ptr : constant Func_Ptr := Func'Access;
      |
   >>> warning: "Access" attribute of "Func" before body seen
   >>> warning: possible Program_Error on later references
   >>> warning:   body of unit "Attribute_Call" elaborated
   >>> warning:   "Access" of "Func" taken at line 5

6.
7.   function Func return Integer is
8.   begin
9.     ...
10.  end Func;
```

```
11. end Attribute_Call;
```

In the example above, the elaboration of declaration `Ptr` is assigned `Func'Access` before the body of `Func` has been elaborated.

`-gnatwl`

Turn on warnings for elaboration problems

When this switch is in effect, GNAT emits diagnostics in the form of warnings concerning various elaboration problems. The warnings are enabled by default. The switch is provided in case all warnings are suppressed, but elaboration warnings are still desired.

`-gnatwL`

Turn off warnings for elaboration problems

When this switch is in effect, GNAT no longer emits any diagnostics in the form of warnings. Selective suppression of elaboration problems is possible using `pragma Warnings (Off)`.

```
1. package body Selective_Suppression is
2.   function ABE return Integer;
3.
4.   Val_1 : constant Integer := ABE;
                                   |
   >>> warning: cannot call "ABE" before body seen
   >>> warning: Program_Error will be raised at run time

5.
6.   pragma Warnings (Off);
7.   Val_2 : constant Integer := ABE;
8.   pragma Warnings (On);
9.
10.  function ABE return Integer is
11.  begin
12.    ...
13.  end ABE;
14. end Selective_Suppression;
```

Note that suppressing elaboration warnings does not eliminate run-time checks. The example above will still fail at run time with an ABE.

## 9.12 Summary of Procedures for Elaboration Control

You should first compile the program with the default options, using none of the binder or compiler switches. If the binder succeeds in finding an elaboration order, then apart from possible cases involving dispatching calls and access-to-subprogram types, the program is free of elaboration errors.

If it is important for the program to be portable to compilers other than GNAT, you should use compiler switch `-gnatel` and consider the messages about missing or implicitly created `Elaborate` and `Elaborate_All` pragmas.

If the binder reports an elaboration circularity, you have several options:

- \* Ensure that elaboration warnings are enabled. This allows the static model to output trace information of elaboration issues. The trace information could shed light on previously unforeseen dependencies as well as their origins. You enable elaboration warnings with compiler switch `-gnatw1`.
- \* Consider the tactics given in the suggestions section of the circularity diagnostic.
- \* If none of the steps outlined above resolve the circularity, use a more permissive elaboration model, in the following order:
  - Use the pre-20.x legacy elaboration-order model, with binder switch `-H`.
  - Use both pre-18.x and pre-20.x legacy elaboration models, with compiler switch `-gnatH` and binder switch `-H`.
  - Use the relaxed static elaboration model, with compiler switches `-gnatH -gnatJ` and binder switch `-H`.
  - Use the relaxed dynamic elaboration model, with compiler switches `-gnatH -gnatJ -gnatE` and binder switch `-H`.

### 9.13 Inspecting the Chosen Elaboration Order

To see the elaboration order chosen by the binder, inspect the contents of file `b~xxx.adb`. On certain targets, this file appears as `b.xxx.adb`. The elaboration order appears as a sequence of calls to `Elab_Body` and `Elab_Spec`, interspersed with assignments to `Exxx` which indicates that a particular unit is elaborated. For example:

```
System.Soft_Links'Elab_Body;
E14 := True;
System.Secondary_Stack'Elab_Body;
E18 := True;
System.Exception_Table'Elab_Body;
E24 := True;
Ada.Io_Exceptions'Elab_Spec;
E67 := True;
Ada.Tags'Elab_Spec;
Ada.Streams'Elab_Spec;
E43 := True;
Interfaces.C'Elab_Spec;
E69 := True;
System.Finalization_Root'Elab_Spec;
E60 := True;
System.Os_Lib'Elab_Body;
E71 := True;
System.Finalization_Implementation'Elab_Spec;
System.Finalization_Implementation'Elab_Body;
E62 := True;
Ada.Finalization'Elab_Spec;
E58 := True;
Ada.Finalization.List_Controller'Elab_Spec;
```

```

E76 := True;
System.File_Control_Block'Elab_Spec;
E74 := True;
System.File_Io'Elab_Body;
E56 := True;
Ada.Tags'Elab_Body;
E45 := True;
Ada.Text_Io'Elab_Spec;
Ada.Text_Io'Elab_Body;
E07 := True;

```

Note also binder switch -l, which outputs the chosen elaboration order and provides a more readable form of the above:

```

ada (spec)
interfaces (spec)
system (spec)
system.case_util (spec)
system.case_util (body)
system.concat_2 (spec)
system.concat_2 (body)
system.concat_3 (spec)
system.concat_3 (body)
system.htable (spec)
system.parameters (spec)
system.parameters (body)
system.crt1 (spec)
interfaces.c_streams (spec)
interfaces.c_streams (body)
system.restrictions (spec)
system.restrictions (body)
system.standard_library (spec)
system.exceptions (spec)
system.exceptions (body)
system.storage_elements (spec)
system.storage_elements (body)
system.secondary_stack (spec)
system.stack_checking (spec)
system.stack_checking (body)
system.string_hash (spec)
system.string_hash (body)
system.htable (body)
system.strings (spec)
system.strings (body)
system.traceback (spec)
system.traceback (body)
system.traceback_entries (spec)
system.traceback_entries (body)

```

```
ada.exceptions (spec)
ada.exceptions.last_chance_handler (spec)
system.soft_links (spec)
system.soft_links (body)
ada.exceptions.last_chance_handler (body)
system.secondary_stack (body)
system.exception_table (spec)
system.exception_table (body)
ada.io_exceptions (spec)
ada.tags (spec)
ada.streams (spec)
interfaces.c (spec)
interfaces.c (body)
system.finalization_root (spec)
system.finalization_root (body)
system.memory (spec)
system.memory (body)
system.standard_library (body)
system.os_lib (spec)
system.os_lib (body)
system.unsigned_types (spec)
system.stream_attributes (spec)
system.stream_attributes (body)
system.finalization_implementation (spec)
system.finalization_implementation (body)
ada.finalization (spec)
ada.finalization (body)
ada.finalization.list_controller (spec)
ada.finalization.list_controller (body)
system.file_control_block (spec)
system.file_io (spec)
system.file_io (body)
system.val_uns (spec)
system.val_util (spec)
system.val_util (body)
system.val_uns (body)
system.wch_con (spec)
system.wch_con (body)
system.wch_cnv (spec)
system.wch_jis (spec)
system.wch_jis (body)
system.wch_cnv (body)
system.wch_stw (spec)
system.wch_stw (body)
ada.tags (body)
ada.exceptions (body)
ada.text_io (spec)
```

```
ada.text_io (body)
text_io (spec)
gdbstr (body)
```

## 10 Inline Assembler

If you need to write low-level software that interacts directly with the hardware, Ada provides two ways for you to incorporate assembly language code into your program. First, you can import and invoke external routines written in assembly language, an Ada feature fully supported by GNAT. However, for small sections of code, it may be simpler or more efficient to include assembly language statements directly in your Ada source program, using the facilities of the implementation-defined package `System.Machine_Code`, which incorporates the gcc Inline Assembler. The Inline Assembler approach offers a number of advantages, including the following:

- \* No need to use non-Ada tools
- \* Consistent interface over different targets
- \* Automatic usage of the proper calling conventions
- \* Access to Ada constants and variables
- \* Definition of intrinsic routines
- \* Possibility of inlining a subprogram consisting of assembler code
- \* Code optimizer can take Inline Assembler code into account

This appendix presents a series of examples to show you how to use the Inline Assembler. Although it focuses on the Intel x86, the general approach applies also to other processors. It is assumed you are familiar with both Ada and assembly language programming.

### 10.1 Basic Assembler Syntax

The assembler used by GNAT and gcc is based not on the Intel assembly language, but rather on a language that descends from the AT&T Unix assembler `as` (and which is often referred to as ‘AT&T syntax’). The following table summarizes the main features of `as` syntax and points out the differences from the Intel conventions. See the gcc `as` and `gas` (an `as` macro pre-processor) documentation for further information.

‘Register names’

gcc / `as`: Prefix with ‘%’; for example `%eax`  
 Intel: No extra punctuation; for example `eax`

‘Immediate operand’

gcc / `as`: Prefix with ‘\$’; for example `$4`  
 Intel: No extra punctuation; for example `4`

‘Address’

gcc / `as`: Prefix with ‘\$’; for example `$loc`  
 Intel: No extra punctuation; for example `loc`

‘Memory contents’

gcc / `as`: No extra punctuation; for example `loc`  
 Intel: Square brackets; for example `[loc]`

‘Register contents’

gcc / `as`: Parentheses; for example `(%eax)`  
 Intel: Square brackets; for example `[eax]`

‘Hexadecimal numbers’

gcc / **as**: Leading ‘0x’ (C language syntax); for example 0xA0  
 Intel: Trailing ‘h’; for example A0h

‘Operand size’  
 gcc / **as**: Explicit in op code; for example `movw` to move a 16-bit word  
 Intel: Implicit, deduced by assembler; for example `mov`

‘Instruction repetition’  
 gcc / **as**: Split into two lines; for example  
     `rep`  
     `stosl`  
 Intel: Keep on one line; for example `rep stosl`

‘Order of operands’  
 gcc / **as**: Source first; for example `movw $4, %eax`  
 Intel: Destination first; for example `mov eax, 4`

## 10.2 A Simple Example of Inline Assembler

The following example generate a single assembly language statement, `nop`, which does nothing. Despite its lack of run-time effect, the example is useful in illustrating the basics of the Inline Assembler facility.

```
with System.Machine_Code; use System.Machine_Code;
procedure Nothing is
begin
  Asm ("nop");
end Nothing;
```

`Asm` is a procedure declared in package `System.Machine_Code`; here it takes one parameter, a ‘template string’ that must be a static expression that produces the generated instruction. `Asm` may be regarded as a compile-time procedure that parses the template string and any additional parameters (none, in this case) and generates one or more assembly language instructions.

The examples in this chapter will illustrate several of the forms for invoking `Asm`; a complete specification of the syntax is found in the `Machine_Code_Insertions` section of the *GNAT Reference Manual*.

Under the standard GNAT conventions, you should put the `Nothing` procedure in a file named `nothing.adb`. You can build the executable in the usual way:

```
$ gnatmake nothing
```

However, the interesting aspect of this example is not its run-time behavior but rather the generated assembly code. To see this output, invoke the compiler as follows:

```
$ gcc -c -S -fomit-frame-pointer -gnatp nothing.adb
```

where the options are:

```
*
-c
    compile only (no bind or link)
*
-S
```

```

        generate assembler listing
*
-fomit-frame-pointer
        do not set up separate stack frames
*
-gnatp
        do not add runtime checks

```

This gives a human-readable assembler version of the code. The resulting file has the same name as the Ada source file but with a `.s` extension. In our example, the file `nothing.s` has the following contents:

```

.file "nothing.adb"
gcc2_compiled.:
__gnu_compiled_ada:
.text
    .align 4
.globl __ada_nothing
__ada_nothing:
#APP
    nop
#NO_APP
    jmp L1
    .align 2,0x90
L1:
    ret

```

The assembly code you included is clearly indicated by the compiler, between the `#APP` and `#NO_APP` delimiters. The character before the ‘APP’ and ‘NOAPP’ can differ on different targets. For example, GNU/Linux uses ‘#APP’ while on NT you will see ‘/APP’.

If you make a mistake in your assembler code (such as using the wrong size modifier or using a wrong operand for the instruction) GNAT will report this error in a temporary file, which is deleted when the compilation is finished. Generating an assembler file will help in such cases, since you can assemble this file separately using the `as` assembler that comes with gcc.

Assembling the file using the command

```
$ as nothing.s
```

will give you error messages whose lines correspond to the assembler input file, so you can easily find and correct any mistakes you made. If there are no errors, `as` generates an object file called `nothing.out`.

### 10.3 Output Variables in Inline Assembler

The examples in this section, showing how to access the processor flags, illustrate how to specify the destination operands for assembly language statements.

```

with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;

```

```

with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags is
  Flags : Unsigned_32;
  use ASCII;
begin
  Asm ("pushfl"           & LF & HT & -- push flags on stack
      "popl %%eax"        & LF & HT & -- load eax with flags
      "movl %%eax, %0",    -- store flags in variable
      Outputs => Unsigned_32'Asm_Output ("=g", Flags));
  Put_Line ("Flags register:" & Flags'Img);
end Get_Flags;

```

We have separated multiple assembler statements in the Asm template string with linefeed (ASCII.LF) and horizontal tab (ASCII.HT) characters in order to have a nicely aligned assembly listing. The resulting section of the assembly output file is:

```

#APP
    pushfl
    popl %eax
    movl %eax, -40(%ebp)
#NO_APP

```

It would have been legal to write the Asm invocation as:

```
Asm ("pushfl popl %%eax movl %%eax, %0")
```

but in the generated assembler file, this would come out as:

```

#APP
    pushfl popl %eax movl %eax, -40(%ebp)
#NO_APP

```

which is not so convenient for the human reader.

We use Ada comments at the end of each line to explain what the assembler instructions actually do. This is a useful convention.

When writing Inline Assembler instructions, you need to precede each register and variable name with a percent sign. Since the assembler already requires a percent sign at the beginning of a register name, you need two consecutive percent signs for such names in the Asm template string, thus `%%eax`. In the generated assembly code, one of the percent signs will be stripped off.

Names such as `%0`, `%1`, `%2`, etc., denote input or output variables: operands you later define using `Input` or `Output` parameters to `Asm`. An output variable is shown in the third section of the Asm template string:

```
movl %%eax, %0
```

The intent of this section is to store the contents of the `eax` register in a variable that can be accessed in Ada. Simply writing `movl %%eax, Flags` would not necessarily work, since the compiler might optimize by using a register to hold `Flags`, and the expansion of the `movl` instruction would not be aware of this optimization. The solution is not to store the result directly but rather to advise the compiler to choose the correct operand form; that is the purpose of the `%0` output variable.

Information about the output variable is supplied in the `Outputs` parameter to `Asm`:

```
Outputs => Unsigned_32'Asm_Output ("=g", Flags));
```

The output is defined by the `Asm_Output` attribute of the target type; the general format is:

```
Type'Asm_Output (constraint_string, variable_name)
```

The constraint string directs the compiler how to store/access the associated variable. In the example:

```
Unsigned_32'Asm_Output ("=m", Flags);
```

the `"m"` (memory) constraint tells the compiler that the variable `Flags` should be stored in a memory variable, thus preventing the optimizer from keeping it in a register. In contrast,

```
Unsigned_32'Asm_Output ("=r", Flags);
```

uses the `"r"` (register) constraint, telling the compiler to store the variable in a register.

If you precede the constraint with the equal character (`'='`), it tells the compiler that the variable will have data stored into it.

In the `Get_Flags` example, we used the `"g"` (global) constraint, allowing the optimizer to choose whatever operand it deems best.

There are a fairly large number of constraints, but the ones that are most useful for the Intel x86 processor are the following:

<code>'='</code>	output constraint
<code>'g'</code>	global (i.e., can be stored anywhere)
<code>'m'</code>	in memory
<code>'I'</code>	a constant
<code>'a'</code>	use <code>eax</code>
<code>'b'</code>	use <code>ebx</code>
<code>'c'</code>	use <code>ecx</code>
<code>'d'</code>	use <code>edx</code>
<code>'S'</code>	use <code>esi</code>
<code>'D'</code>	use <code>edi</code>
<code>'r'</code>	use one of <code>eax</code> , <code>ebx</code> , <code>ecx</code> or <code>edx</code>
<code>'q'</code>	use one of <code>eax</code> , <code>ebx</code> , <code>ecx</code> , <code>edx</code> , <code>esi</code> or <code>edi</code>

The full set of constraints is described in the `gcc` and `as` documentation; note that you can combine certain constraints into one constraint string.

You specify the association of an output variable with an assembler operand through the `%n` notation, where 'n' is a non-negative integer. Thus in

```
Asm ("pushfl"          & LF & HT & -- push flags on stack
     "popl %%eax"      & LF & HT & -- load eax with flags
     "movl %%eax, %0",  -- store flags in variable
     Outputs => Unsigned_32'Asm_Output ("=g", Flags));
```

`%0` is replaced in the expanded code by the appropriate operand, whatever the compiler chose for the `Flags` variable.

In general, you may have any number of output variables:

- \* Count the operands starting at 0; thus `%0`, `%1`, etc.
- \* Specify the `Outputs` parameter as a parenthesized comma-separated list of `Asm_Output` attributes

For example:

```
Asm ("movl %%eax, %0" & LF & HT &
     "movl %%ebx, %1" & LF & HT &
     "movl %%ecx, %2",
     Outputs => (Unsigned_32'Asm_Output ("=g", Var_A), -- %0 = Var_A
                 Unsigned_32'Asm_Output ("=g", Var_B), -- %1 = Var_B
                 Unsigned_32'Asm_Output ("=g", Var_C))); -- %2 = Var_C
```

where `Var_A`, `Var_B`, and `Var_C` are variables in the Ada program.

As a variation on the `Get_Flags` example, we can use the constraint string to direct the compiler to store the `eax` register into the `Flags` variable, instead of including the store instruction explicitly in the `Asm` template string:

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags_2 is
  Flags : Unsigned_32;
  use ASCII;
begin
  Asm ("pushfl"          & LF & HT & -- push flags on stack
       "popl %%eax",      -- save flags in eax
       Outputs => Unsigned_32'Asm_Output ("=a", Flags));
  Put_Line ("Flags register:" & Flags'Img);
end Get_Flags_2;
```

The `"a"` constraint tells the compiler that the `Flags` variable will come from the `eax` register. Here is the resulting code:

```
#APP
  pushfl
  popl %%eax
#NO_APP
  movl %%eax,-40(%ebp)
```

The compiler generated the store of `eax` into `Flags` after expanding the assembler code. In fact, there was no need to pop the flags into the `eax` register; more simply, we could just pop the flags directly into the program variable:

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags_3 is
  Flags : Unsigned_32;
  use ASCII;
begin
  Asm ("pushfl" & LF & HT & -- push flags on stack
      "pop %0",           -- save flags in Flags
      Outputs => Unsigned_32'Asm_Output ("=g", Flags));
  Put_Line ("Flags register:" & Flags'Img);
end Get_Flags_3;
```

## 10.4 Input Variables in Inline Assembler

The example in this section illustrates how to specify the source operands for assembly language statements. The procedure simply increments its input value by 1:

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Increment is

  function Incr (Value : Unsigned_32) return Unsigned_32 is
    Result : Unsigned_32;
  begin
    Asm ("incl %0",
        Outputs => Unsigned_32'Asm_Output ("=a", Result),
        Inputs  => Unsigned_32'Asm_Input  ("a", Value));
    return Result;
  end Incr;

  Value : Unsigned_32;

begin
  Value := 5;
  Put_Line ("Value before is" & Value'Img);
  Value := Incr (Value);
  Put_Line ("Value after is" & Value'Img);
end Increment;
```

The `Outputs` parameter to `Asm` specifies that the result is in the `eax` register and that it is to be stored in the `Result` variable.

The `Inputs` parameter looks much like the `Outputs` parameter, but with an `Asm_Input` attribute. The `"="` constraint, indicating an output value, is not present.

You can have multiple input variables in the same way you can have more than one output variable.

The parameter count (%0, %1) etc, still starts at the first output statement, and continues with the input statements.

Just as the **Outputs** parameter causes the register to be stored into the target variable after execution of the assembler statements, the **Inputs** parameter causes its variable to be loaded into the register before execution of the assembler statements.

Thus the effect of the **Asm** invocation is:

- \* load the 32-bit value of **Value** into **eax**
- \* execute the **incl %eax** instruction
- \* store the contents of **eax** into the **Result** variable

The resulting assembler file (with -O2 optimization) contains:

```
_increment__incr.1:
    subl $4,%esp
    movl 8(%esp),%eax
#APP
    incl %eax
#NO_APP
    movl %eax,%edx
    movl %ecx, (%esp)
    addl $4,%esp
    ret
```

## 10.5 Inlining Inline Assembler Code

For a short subprogram such as the **Incr** function in the previous section, the overhead of the call and return (creating / deleting the stack frame) can be significant, compared to the amount of code in the subprogram body. A solution is to apply Ada's **Inline** pragma to the subprogram, which directs the compiler to expand invocations of the subprogram at the point(s) of call, instead of setting up a stack frame for out-of-line calls. Here's the resulting program:

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Increment_2 is

    function Incr (Value : Unsigned_32) return Unsigned_32 is
        Result : Unsigned_32;
    begin
        Asm ("incl %0",
            Outputs => Unsigned_32'Asm_Output ("=a", Result),
            Inputs  => Unsigned_32'Asm_Input ("a", Value));
        return Result;
    end Incr;
pragma Inline (Increment);
```

```

Value : Unsigned_32;

begin
  Value := 5;
  Put_Line ("Value before is" & Value'Img);
  Value := Increment (Value);
  Put_Line ("Value after is" & Value'Img);
end Increment_2;

```

Compile the program with both optimization (`-O2`) and inlining (`-gnatn`) enabled.

The `Incr` function is still compiled as usual, but at the point in `Increment` where our function used to be called:

```

pushl %edi
call _increment__incr.1

```

the code for the function body directly appears:

```

movl %esi,%eax
#APP
  incl %eax
#NO_APP
movl %eax,%edx

```

thus saving the overhead of stack frame setup and an out-of-line call.

## 10.6 Other Asm Functionality

This section describes two important parameters to the `Asm` procedure: `Clobber`, which identifies register usage; and `Volatile`, which inhibits unwanted optimizations.

### 10.6.1 The Clobber Parameter

One of the dangers of intermixing assembly language and a compiled language such as Ada is that the compiler needs to be aware of which registers are being used by the assembly code. In some cases, such as the earlier examples, the constraint string is sufficient to indicate register usage (e.g., `"a"` for the `eax` register). But, more generally, the compiler needs an explicit identification of the registers that are used by the Inline Assembly statements.

Using a register that the compiler doesn't know about could be a side effect of an instruction (like `mull`, which stores its result into both `eax` and `edx`). It can also arise from explicit register usage within your assembly code; for example:

```

Asm ("movl %0, %%ebx" & LF & HT &
    "movl %%ebx, %1",
    Outputs => Unsigned_32'Asm_Output ("g", Var_Out),
    Inputs  => Unsigned_32'Asm_Input  ("g", Var_In));

```

where the compiler (since it does not analyze the `Asm` template string) does not know you are using the `ebx` register.

In such cases you need to supply the `Clobber` parameter to `Asm`, to identify the registers used by your assembly code:

```

Asm ("movl %0, %%ebx" & LF & HT &

```

```

    "movl %%ebx, %1",
    Outputs => Unsigned_32'Asm_Output ("=g", Var_Out),
    Inputs  => Unsigned_32'Asm_Input  ("g", Var_In),
    Clobber => "ebx");

```

The `Clobber` parameter is a static string expression specifying the register(s) you are using. Note that register names are ‘not’ prefixed by a percent sign. Also, if more than one register is used, you separate their names by commas; e.g., `"eax, ebx"`

The `Clobber` parameter has several additional uses:

- \* Use ‘register’ name `cc` to indicate that flags might have changed
- \* Use ‘register’ name `memory` if you changed a memory location

### 10.6.2 The Volatile Parameter

Compiler optimizations in the presence of Inline Assembler may sometimes have unwanted effects. For example, when an `Asm` invocation with an input variable is inside a loop, the compiler might move the loading of the input variable outside the loop, regarding it as a one-time initialization.

If you don’t want this to happen, you can disable such optimizations by setting the `Volatile` parameter to `True`; for example:

```

    Asm ("movl %0, %%ebx" & LF & HT &
        "movl %%ebx, %1",
        Outputs => Unsigned_32'Asm_Output ("=g", Var_Out),
        Inputs  => Unsigned_32'Asm_Input  ("g", Var_In),
        Clobber  => "ebx",
        Volatile => True);

```

By default, `Volatile` is set to `False` unless there is no `Outputs` parameter.

Although setting `Volatile` to `True` prevents unwanted optimizations, it also disables other optimizations that might be important for efficiency. In general, you should set `Volatile` to `True` only if the compiler’s optimizations have created problems.

# 11 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc ‘<https://fsf.org/>’

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

‘Preamble’

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

‘1. APPLICABILITY AND DEFINITIONS’

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ‘Document’, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called ‘Opaque’.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## ‘2. VERBATIM COPYING’

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute.

However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

### ‘3. COPYING IN QUANTITY’

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

### ‘4. MODIFICATIONS’

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes

a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

#### ‘5. COMBINING DOCUMENTS’

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

#### ‘6. COLLECTIONS OF DOCUMENTS’

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

#### ‘7. AGGREGATION WITH INDEPENDENT WORKS’

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

#### ‘8. TRANSLATION’

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations

requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

#### ‘9. TERMINATION’

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

#### ‘10. FUTURE REVISIONS OF THIS LICENSE’

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See ‘<https://www.gnu.org/copyleft/>’.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

#### ‘11. RELICENSING’

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A

“Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

‘ADDENDUM: How to use this License for your documents’

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with . . . Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.





- gnatVo (gcc) ..... 134
- gnatVp (gcc) ..... 134
- gnatVr (gcc) ..... 134
- gnatVs (gcc) ..... 134
- gnatVt (gcc) ..... 134
- gnatw (gcc) ..... 103
- gnatw.a (gcc) ..... 112
- gnatw.A (gcc) ..... 112
- gnatw.b (gcc) ..... 112
- gnatw.c (gcc) ..... 113
- gnatw.C (gcc) ..... 113
- gnatw.d (gcc) ..... 114
- gnatw.e (gcc) ..... 115
- gnatw.f (gnat) ..... 303
- gnatw.g (gcc) ..... 115
- gnatw.h (gcc) ..... 116
- gnatw.H (gcc) ..... 116
- gnatw.i (gcc) ..... 116
- gnatw.I (gcc) ..... 117
- gnatw.j (gcc) ..... 117
- gnatw.J (gcc) ..... 117
- gnatw.k (gcc) ..... 118
- gnatw.l (gcc) ..... 118
- gnatw.L (gcc) ..... 118
- gnatw.m (gcc) ..... 119
- gnatw.M (gcc) ..... 119
- gnatw.n (gcc) ..... 119
- gnatw.N (gcc) ..... 119
- gnatw.o (gcc) ..... 120
- gnatw.O (gcc) ..... 120
- gnatw.p (gcc) ..... 120
- gnatw.P (gcc) ..... 120
- gnatw.q (gcc) ..... 121
- gnatw.Q (gcc) ..... 122
- gnatw.r (gcc) ..... 123
- gnatw.R (gcc) ..... 123
- gnatw.s (gcc) ..... 123
- gnatw.S (gcc) ..... 123
- gnatw.t (gcc) ..... 124
- gnatw.T (gcc) ..... 124
- gnatw.u (gcc) ..... 125
- gnatw.U (gcc) ..... 125
- gnatw.v (gcc) ..... 125
- gnatw.V (gcc) ..... 126
- gnatw.w (gcc) ..... 126
- gnatw.W (gcc) ..... 126
- gnatw.x (gcc) ..... 127
- gnatw.y (gcc) ..... 127
- gnatw.Y (gcc) ..... 127
- gnatw.z (gcc) ..... 128
- gnatw.Z (gcc) ..... 128
- gnatw\_a ..... 112
- gnatw\_A ..... 112
- gnatw\_c (gcc) ..... 113
- gnatw\_C (gcc) ..... 114
- gnatw\_l (gcc) ..... 118
- gnatw\_L (gcc) ..... 118
- gnatw\_p (gcc) ..... 121
- gnatw\_P (gcc) ..... 121
- gnatw\_q (gcc) ..... 122
- gnatw\_Q (gcc) ..... 122
- gnatw\_r (gcc) ..... 123
- gnatw\_R (gcc) ..... 123
- gnatw\_s (gcc) ..... 124
- gnatw\_S (gcc) ..... 124
- gnatW (gcc) ..... 103, 147
- gnatwa (gcc) ..... 111
- gnatwA (gcc) ..... 111
- gnatwb (gcc) ..... 112
- gnatwB (gcc) ..... 112, 113
- gnatwc (gcc) ..... 113
- gnatwC (gcc) ..... 113
- gnatwd (gcc) ..... 114
- gnatwD (gcc) ..... 114
- gnatwe (gcc) ..... 115
- gnatwE (gcc) ..... 115
- gnatwf (gcc) ..... 115
- gnatwF (gcc) ..... 115
- gnatwg (gcc) ..... 115
- gnatwG (gcc) ..... 115
- gnatwh (gcc) ..... 116
- gnatwH (gcc) ..... 116
- gnatwi (gcc) ..... 116
- gnatwI (gcc) ..... 116
- gnatwj (gcc) ..... 117
- gnatwJ (gcc) ..... 117
- gnatwk (gcc) ..... 117
- gnatwK (gcc) ..... 117, 118
- gnatwl (gcc) ..... 118
- gnatwl (gnat) ..... 304
- gnatwL (gcc) ..... 118
- gnatwm (gcc) ..... 118
- gnatwM (gcc) ..... 119
- gnatwn (gcc) ..... 119
- gnatwo (gcc) ..... 119
- gnatwO (gcc) ..... 120
- gnatwp (gcc) ..... 120
- gnatwP (gcc) ..... 120
- gnatwq (gcc) ..... 121
- gnatwQ (gcc) ..... 121
- gnatwr (gcc) ..... 122
- gnatwR (gcc) ..... 123
- gnatws (gcc) ..... 123
- gnatwt (gcc) ..... 124
- gnatwT (gcc) ..... 124
- gnatwu (gcc) ..... 124
- gnatwU (gcc) ..... 125
- gnatwv (gcc) ..... 125
- gnatwV (gcc) ..... 125
- gnatww (gcc) ..... 126
- gnatwW (gcc) ..... 126
- gnatwx (gcc) ..... 126
- gnatwX (gcc) ..... 126
- gnatwy (gcc) ..... 127
- gnatwY (gcc) ..... 127
- gnatwz (gcc) ..... 127

- gnatwZ (gcc)..... 128
- gnatx (gcc)..... 103, 153
- gnatX (gcc)..... 103, 146
- gnatX0 (gcc)..... 103, 146
- gnaty (gcc)..... 103, 135
- gnaty+ (gcc)..... 142
- gnaty- (gcc)..... 142
- gnaty[0-9] (gcc)..... 135
- gnatya (gcc)..... 135
- gnatyA (gcc)..... 136
- gnatyb (gcc)..... 136
- gnatyB (gcc)..... 136
- gnatyc (gcc)..... 136
- gnatyC (gcc)..... 137
- gnatyd (gcc)..... 137
- gnatyD (gcc)..... 137
- gnatye (gcc)..... 137
- gnatyf (gcc)..... 137
- gnatyg (gcc)..... 137
- gnatyh (gcc)..... 137
- gnatyi (gcc)..... 137
- gnatyI (gcc)..... 138
- gnatyk (gcc)..... 138
- gnatyl (gcc)..... 138
- gnatyLnnn (gcc)..... 139
- gnatym (gcc)..... 139
- gnatyMnnn (gcc)..... 139
- gnatyn (gcc)..... 139
- gnatyN (gcc)..... 140
- gnatyo (gcc)..... 140
- gnatyO (gcc)..... 140
- gnatyp (gcc)..... 140
- gnatyr (gcc)..... 140
- gnatys (gcc)..... 140
- gnatyS (gcc)..... 140
- gnatyt (gcc)..... 141
- gnatyu (gcc)..... 141
- gnatyx (gcc)..... 141
- gnatyy (gcc)..... 141
- gnatyz (gcc)..... 142
- gnatz (gcc)..... 103
- h (gnatbind)..... 158, 165
- h (gnatclean)..... 179
- h (gnatdll)..... 261
- h (gnatls)..... 181
- h (gnatname)..... 17
- H (gnatbind)..... 158
- H32 (gnatbind)..... 158
- H64 (gnatbind)..... 158
- i (gnatmake)..... 81
- I (gcc)..... 104
- I (gnatbind)..... 158, 168
- I (gnatclean)..... 180
- I (gnatdll)..... 261
- I (gnatls)..... 181
- I (gnatmake)..... 85
- I- (gcc)..... 104
- I- (gnatbind)..... 159
- I- (gnatclean)..... 180
- I- (gnatls)..... 181
- I- (gnatmake)..... 85
- j (gnatmake)..... 82
- k (dlltool)..... 263
- k (gnatbind)..... 159
- k (gnatchop)..... 23
- k (gnatdll)..... 261
- k (gnatmake)..... 82
- K (gnatbind)..... 159, 165
- l (gnatbind)..... 159, 165
- l (gnatdll)..... 261
- l (gnatmake)..... 82
- L (gnatbind)..... 159
- L (gnatmake)..... 85
- largs (gnatdll)..... 261
- largs (gnatmake)..... 86
- m (gnatbind)..... 159, 163
- m (gnatmake)..... 82
- M (gnatbind)..... 159, 163
- M (gnatlink)..... 171
- M (gnatmake)..... 82
- M= (gnatlink)..... 171
- margs (gnatmake)..... 86
- minimal (gnatbind)..... 159
- mwindows..... 246
- n (gnatbind)..... 159, 166, 167
- n (gnatclean)..... 179
- n (gnatdll)..... 261
- n (gnatlink)..... 170
- n (gnatmake)..... 83
- nostdinc (gcc)..... 104
- nostdinc (gnatbind)..... 159
- nostdinc (gnatmake)..... 85
- nostdlib (gcc)..... 104
- nostdlib (gnatbind)..... 160
- nostdlib (gnatmake)..... 85
- o (gcc)..... 104
- o (gnatbind)..... 160, 166, 167
- o (gnatlink)..... 170
- o (gnatls)..... 181
- o (gnatmake)..... 83
- O (gcc)..... 104, 206
- O (gnatbind)..... 160, 165
- p (gnatbind)..... 160, 165
- p (gnatchop)..... 23
- p (gnatmake)..... 83
- P (gnatbind)..... 160
- P (gnatclean)..... 179
- P (gnatmake)..... 78
- P (gnatname)..... 17
- pass-exit-codes (gcc)..... 104, 149
- pg (gcc)..... 203
- pg (gnatlink)..... 203
- q (gnatchop)..... 23
- q (gnatclean)..... 179
- q (gnatdll)..... 261
- q (gnatmake)..... 83

- Q (gnatbind) ..... 160
  - r (gnatbind) ..... 166
  - r (gnatchop) ..... 23
  - r (gnatclean) ..... 179
  - r (gnatprep) ..... 46
  - R (gnatbind) ..... 160
  - Ra (gnatbind) ..... 160
  - s (gnatbind) ..... 160, 162
  - s (gnatls) ..... 181
  - s (gnatmake) ..... 83
  - s (gnatprep) ..... 46
  - S (gcc) ..... 105
  - S (gnatbind) ..... 160
  - shared (gnatbind) ..... 161
  - static (gnatbind) ..... 161
  - t (gnatbind) ..... 161, 163
  - T (gnatbind) ..... 161
  - T (gnatprep) ..... 46
  - T0 switch ..... 244
  - u (gnatbind) ..... 161
  - u (gnatls) ..... 181
  - u (gnatmake) ..... 83
  - u (gnatprep) ..... 46
  - U (gnatmake) ..... 83
  - v (dlltool) ..... 263
  - v (gcc) ..... 105
  - v (gnatbind) ..... 162, 163
  - v (gnatchop) ..... 24
  - v (gnatclean) ..... 179
  - v (gnatdll) ..... 261
  - v (gnatlink) ..... 170
  - v (gnatls) ..... 182
  - v (gnatmake) ..... 83
  - v (gnatname) ..... 17
  - v (gnatprep) ..... 46
  - v -v (gnatlink) ..... 170
  - v -v (gnatname) ..... 17
  - V (gcc) ..... 105
  - V (gnatbind) ..... 162
  - vl (gnatmake) ..... 84
  - vm (gnatmake) ..... 84
  - vP (gnatclean) ..... 179
  - w (gcc) ..... 105, 128
  - w (gnatbind) ..... 162
  - w (gnatchop) ..... 24
  - Wall (gcc) ..... 128
  - we (gnatbind) ..... 163
  - Werror (gcc) ..... 128
  - ws (gnatbind) ..... 163
  - Wstack-usage (gcc) ..... 128
  - Wuninitialized (gcc) ..... 128
  - Wunused (gcc) ..... 128
  - Wx (gnatbind) ..... 162
  - x (gnatbind) ..... 162, 163
  - x (gnatmake) ..... 84
  - x (gnatname) ..... 17
  - X (gnatclean) ..... 179
  - xdr (gnatbind) ..... 162
  - Xnnn (gnatbind) ..... 162
  - y (gnatbind) ..... 162
  - z (gnatbind) ..... 162, 167
  - z (gnatmake) ..... 84
  - 
  - gnat\_malloc ..... 158
- ## A
- Abnormal Termination or
    - Failure to Terminate ..... 192
  - access before elaboration ..... 142
  - Access before elaboration ..... 142
  - activate every optional warning ..... 115
  - ACVC ..... 145
  - Ada ..... 168
  - Ada 2005 Language Reference Manual ..... 3
  - Ada 2005 mode ..... 145
  - Ada 2012 mode ..... 145
  - Ada 2022 mode ..... 146
  - Ada 83 mode ..... 145
  - Ada 83 tests ..... 145
  - Ada 95 Language Reference Manual ..... 3
  - Ada 95 mode ..... 145
  - Ada compatibility issues warnings ..... 127
  - Ada expressions (in gdb) ..... 188
  - Ada language extensions ..... 146
  - Ada Library Information files ..... 29
  - Ada.Characters.Latin\_1 ..... 8
  - ADA\_INCLUDE\_PATH ..... 35, 89
  - ADA\_OBJECTS\_PATH ..... 35, 168
  - ADA\_PRJ\_INCLUDE\_FILE ..... 89
  - ADA\_PRJ\_OBJECTS\_FILE ..... 168
  - ADA\_PROJECT\_PATH ..... 33
  - adafinal ..... 167
  - adainit ..... 166
  - Address Clauses ..... 119
  - AddressSanitizer ..... 236
  - ALI files ..... 29
  - Aliasing ..... 213, 217
  - alternative ..... 13
  - Annex A (in Ada Reference Manual) ..... 193
  - Annex B (in Ada reference Manual) ..... 193
  - Anonymous allocators ..... 112
  - APIENTRY ..... 249
  - ASan ..... 236
  - Asm ..... 55
  - Assert ..... 130
  - Assert failures ..... 112
  - Assertions ..... 130
  - Atomic ..... 218
  - Atomic Synchronization ..... 119
  - attach to process ..... 267

**B**

Bad fixed values ..... 112  
 Biased representation ..... 112  
 Binder ..... 167  
 Binder consistency checks ..... 163  
 Binder output (example) ..... 271  
 Binder output file ..... 53  
 Binding generation (for Ada specs) ..... 72  
 Binding generation (for C and C++ headers) .... 69  
 BINUTILS\_ROOT ..... 58  
 bit order warnings ..... 125  
 Breakpoints and tasks ..... 190  
 building ..... 255, 256, 264  
 Building the GNAT Run-Time Library ..... 39

**C**

C headers (binding generation) ..... 69, 72  
 C varargs function ..... 55  
 C++ ..... 56  
 C++ headers (binding generation) ..... 69  
 C\_INCLUDE\_PATH ..... 58  
 Calling Conventions ..... 54  
 cannot generate code ..... 88  
 Check ..... 143  
 Checks ..... 142, 143, 205  
 Checks (overflow) ..... 221  
 COBOL ..... 55  
   code page 437 (IBM PC) ..... 8  
   code page 850 (IBM PC) ..... 9  
 Combining GNAT switches ..... 105  
 Command Line Argument Expansion ..... 247  
 Command line length ..... 170  
 Compatibility with Ada 83 ..... 145  
 compilation (definition) ..... 8  
 Compilation model ..... 7  
 Compile\_Time\_Error ..... 113  
 Compile\_Time\_Warning ..... 113  
 compiling ..... 264  
 Component clause ..... 113  
 Conditional compilation ..... 39  
 Conditionals ..... 113  
 configuration ..... 25  
 Configuration pragmas ..... 25  
 Consistency checks ..... 163  
 CONSOLE Subsystem ..... 246  
 constant ..... 113  
 C ..... 55  
 Convention Ada ..... 54  
 Convention Asm ..... 55  
 Convention Assembler ..... 55  
 Convention C ..... 55  
 Convention C++ ..... 56  
 Convention COBOL ..... 55  
 Convention Default ..... 56  
 Convention DLL ..... 57  
 Convention External ..... 56  
 Convention Fortran ..... 56

Convention Stdcall ..... 57  
 Convention Stubbed ..... 57  
 Convention Win32 ..... 57  
 Conventions ..... 3  
 CR ..... 7  
 Cyrillic ..... 8

**D**

Deactivated code ..... 124  
 Debug ..... 130  
 Debug Pool ..... 233  
 Debugger ..... 184  
 Debugging ..... 184  
 Debugging Generic Units ..... 190  
 Debugging information ..... 170  
 Debugging optimized code ..... 207  
 Debugging options ..... 149  
 Default ..... 56  
 Definition file ..... 253  
 Deleted code ..... 124  
 Dependencies ..... 82  
 Dependency rules (compilation) ..... 77  
 Dereferencing ..... 114  
 Dimension aspect ..... 226, 228  
 Dimension Vector (for a  
   dimensioned subtype) ..... 228  
 Dimension\_System aspect ..... 226, 228  
 Dimensionable type ..... 228  
 Dimensionality analysis ..... 226  
 Dimensioned subtype ..... 228  
 Division by zero ..... 142  
 division by zero ..... 142  
 DLL ..... 57, 251  
 DLL debugging ..... 265, 267  
 DLLs ..... 255, 256  
 DLLs and elaboration ..... 258  
 DLLs and finalization ..... 259  
 Dynamic elaboration model ..... 295

**E**

elaboration ..... 143  
 Elaboration ..... 118  
 Elaboration checks ..... 143  
 Elaboration control ..... 287  
 Elaboration order control ..... 74  
 End of source file; Source file, end ..... 7  
 environment variable;  
   ADA\_INCLUDE\_PATH ..... 35, 89  
 environment variable;  
   ADA\_OBJECTS\_PATH ..... 35, 168  
 environment variable;  
   ADA\_PRJ\_INCLUDE\_FILE ..... 89  
 environment variable;  
   ADA\_PRJ\_OBJECTS\_FILE ..... 168  
 environment variable; BINUTILS\_ROOT ..... 58  
 environment variable; C\_INCLUDE\_PATH ..... 58

environment variable; GCC\_EXEC\_PREFIX ... 58  
 environment variable; GCC\_ROOT ..... 58  
 environment variable; PATH ..... 58, 89, 168  
 environment variable; TMP ..... 246  
 Error messages ..... 108  
 EUC Coding ..... 10  
 Exceptions (in gdb) ..... 189  
 Export table ..... 257  
 Export/Import pragma warnings ..... 126  
 External ..... 56

## F

Features ..... 117  
 FF ..... 7  
 File cleanup tool ..... 178  
 File names ..... 12, 13  
 File Naming Conventions ..... 15  
 File naming schemes ..... 13  
 Fixed-point Small value ..... 112  
 Floating-Point Operations ..... 210  
 for gnatmake ..... 85  
 for profiling ..... 203  
 Foreign Languages ..... 54  
 Formals ..... 115  
 Fortran ..... 56

## G

GCC\_EXEC\_PREFIX ..... 58  
 GCC\_ROOT ..... 58  
 gdb ..... 184  
 Generic formal parameters ..... 145  
 Generics ..... 28, 190  
 gnat.adc ..... 13, 27  
 gnat\_argc ..... 167  
 gnat\_argv ..... 167  
 gnat1 ..... 88  
 gnatbind ..... 155  
 gnatchop ..... 20  
 gnatclean ..... 178  
 gnatdll ..... 260  
 gnatkr ..... 18  
 gnatlink ..... 169  
 gnatls ..... 180  
 gnatmake ..... 77  
 gnatname ..... 15  
 gnatprep ..... 44  
 GNAT ..... 168  
 GNAT (package) ..... 193  
 GNAT compilation model ..... 7  
 GNAT extensions ..... 146  
 GNAT library ..... 74  
 GNAT Run-Time Library ..... 39  
 GNAT\_INIT\_SCALARS ..... 161  
 GNU make ..... 171  
 GNU/Linux ..... 243  
 GPR\_PROJECT\_PATH ..... 33

gprof ..... 202

## H

Hiding of Declarations ..... 116  
 HT ..... 7

## I

implicit ..... 114  
 Implicit dereferencing ..... 114  
 Import library ..... 253  
 Improving performance ..... 204  
 in binder ..... 163  
 including ..... 170  
 Inline ..... 29, 209  
 Inline Assembler ..... 309  
 Inlining ..... 75, 120  
 Interfaces ..... 168  
 Interfacing to Ada ..... 54  
 Interfacing to Assembly ..... 55  
 Interfacing to C ..... 55  
 Interfacing to C varargs function ..... 55  
 Interfacing to C++ ..... 56  
 Interfacing to COBOL ..... 55  
 Interfacing to Fortran ..... 56  
 ISO 8859-15 ..... 8  
 ISO 8859-2 ..... 8  
 ISO 8859-3 ..... 8  
 ISO 8859-4 ..... 8  
 ISO 8859-5 ..... 8

## L

Latin-1 ..... 7, 8  
 Latin-2 ..... 8  
 Latin-3 ..... 8  
 Latin-4 ..... 8  
 Latin-9 ..... 8  
 Layout ..... 121  
 Legacy elaboration models ..... 296  
 LF ..... 7  
 Library browser ..... 180  
 Library building and referencing ..... 31  
 Linker libraries ..... 85  
 Linux ..... 243

## M

make (GNU) ..... 171  
 memory corruption ..... 233  
 Memory Pool ..... 232  
 Microsoft Visual Studio ..... 265  
 missing ..... 113  
 Mixed Language Programming ..... 51  
 MKS\_Type type ..... 226  
 multiple input files ..... 167  
 Multiple units ..... 144

**N**

naming scheme ..... 81  
 No information messages for why  
   package spec needs body ..... 127  
 No\_Strict\_Aliasing ..... 213  
 non-symbolic ..... 194

**O**

obsolescent ..... 117  
 Obsolescent features ..... 117  
 Optimization and debugging ..... 207  
 Optimization Switches ..... 211, 213  
 Order of elaboration ..... 287  
 OS X ..... 269  
 Other Ada compilers ..... 54  
 overflow ..... 143, 205  
 Overflow checks ..... 143, 205, 221  
 Overflow mode ..... 143

**P**

Package spec needing body ..... 127  
 Parallel make ..... 82  
 Parameter order ..... 120  
 Parentheses ..... 121  
 Passive Task ..... 219  
 PATH ..... 58, 89, 168  
 pool ..... 232, 233  
 Postcondition ..... 130  
 pragma Assert ..... 40  
 pragma Assertion\_Policy ..... 40  
 pragma Debug ..... 41  
 pragma Debug\_Policy ..... 41  
 pragma Elaborate (Unit) ..... 292  
 pragma Elaborate\_All (Unit) ..... 293  
 pragma Elaborate\_Body ..... 291  
 pragma Export ..... 76  
 pragma Inline ..... 209  
 pragma Overflow\_Mode ..... 224  
 pragma Preelaborate ..... 291  
 pragma Pure ..... 291  
 pragma Restrictions ..... 151  
 pragma Suppress ..... 205  
 pragma Task\_Dispatching\_Policy ..... 244  
 pragma Time\_Slice ..... 244  
 pragma Unsuppress ..... 205  
 Pragmas ..... 25, 115  
 Precondition ..... 130  
 Preprocessing ..... 44  
 Preprocessing (gnatprep) ..... 44  
 Preprocessors (contrasted with  
   conditional compilation) ..... 40  
 producing list ..... 82  
 Profiling ..... 202

**R**

rc ..... 264  
 rebuilding ..... 39  
 Rebuilding the GNAT Run-Time Library ..... 39  
 Recompilation (by gnatmake) ..... 86  
 Record Representation (component sizes) ..... 123  
 Record Representation (gaps) ..... 116  
 Relaxed elaboration mode ..... 296  
 Remote Debugging with gdbserver ..... 191  
 Resources ..... 263, 264, 265  
 RTL ..... 104  
 Run-time libraries  
   (platform-specific information) ..... 242  
 Run-Time Library ..... 39

**S**

s-digemk.ads file ..... 227  
 Sanitizers ..... 236  
 SCHED\_FIFO scheduling policy ..... 244  
 SCHED\_OTHER scheduling policy ..... 244  
 SCHED\_RR scheduling policy ..... 244  
 Search paths ..... 85  
 setjmp/longjmp Exception Model ..... 242  
 Shift JIS Coding ..... 9  
 Size/Alignment warnings ..... 128  
 SJLJ (setjmp/longjmp Exception Model) ..... 242  
 Small value ..... 112  
 Source files ..... 85, 156, 180  
 Source\_File\_Name pragma ..... 12, 13  
 Source\_Reference pragmas ..... 23  
 SPARK elaboration model ..... 296  
 spec (definition) ..... 8  
 Stack Overflow Checking ..... 143, 231  
 stack overflow checking ..... 142, 143  
 stack traceback ..... 194  
 stack unwinding ..... 194  
 Stand-alone libraries ..... 35  
 Static elaboration model ..... 295  
 Static Stack Usage Analysis ..... 231  
 Stdcall ..... 57, 249  
 stderr ..... 106  
 storage ..... 232, 233  
 Strict Aliasing ..... 213  
 String indexing warnings ..... 126  
 Stubbed ..... 57  
 Style checking ..... 135  
 SUB (control character) ..... 7  
 Subtype predicates ..... 130  
 Subunits ..... 28  
 Subunits (and conditional compilation) ..... 42  
 Suppress ..... 143, 205  
 suppressing ..... 108, 142, 143  
 Suppressing checks ..... 142, 143  
 suppressing search ..... 85, 180  
 symbolic ..... 199  
 symbolic links ..... 81  
 syntax checking ..... 144

System ..... 168  
 System (package in Ada Reference Manual) ... 193  
 System.Dim.Mks package (GNAT library) ..... 226  
 System.IO ..... 89

## T

Task switching (in gdb) ..... 190  
 Tasking and threads libraries ..... 242  
 Tasks (in gdb) ..... 190  
 Temporary files ..... 246  
 Text\_IO and performance ..... 219  
 Threads libraries and tasking ..... 242  
 Time stamp checks ..... 163  
 TMP ..... 246  
 traceback ..... 194, 199  
 treat as error ..... 115  
 Type invariants ..... 130  
 typographical ..... 3  
 Typographical conventions ..... 3

## U

UBSan ..... 238  
 Unassigned variable warnings ..... 125  
 Unchecked\_Conversion warnings ..... 127  
 UndefinedBehaviorSanitizer ..... 238  
 unrecognized ..... 115  
 unreferenced ..... 115  
 Unsuppress ..... 143, 205

Upper-Half Coding ..... 9  
 use by binder ..... 156  
 use with GNAT DLLs ..... 265  
 using ..... 265  
 Unused subprogram/data elimination ..... 220

## V

Validity Checking ..... 131  
 varargs function interfaces ..... 55  
 Version skew (avoided by “gnatmake”) ..... 5  
 Volatile parameter ..... 318  
 VT ..... 7

## W

Warning messages ..... 109  
 warnings ..... 118, 119, 120, 121, 124  
 Warnings ..... 115, 124, 163  
 Warnings Off control ..... 126  
 Win32 ..... 57  
 Windows ..... 245  
 windows ..... 263  
 WINDOWS Subsystem ..... 246  
 windres ..... 264

## Z

ZCX (Zero-Cost Exceptions) ..... 242  
 Zero Cost Exceptions ..... 154  
 Zero-Cost Exceptions ..... 242