GNAT Pro Common Code Generator User's Guide Supplement

Release 27.0w

Apr 28, 2026

This page is intentionally left blank.

GNAT, The GNU Ada Development Environment

GNAT Pro Edition

Version 27.0w
Date: Apr 28, 2026

AdaCore

This page is intentionally left blank.

CONTENTS

1 Getting Started **7**
1.1 Introduction . . . 7
1.2 Calling and Using CCG . . . 7

2 Supported Constructs, Libraries, and Needed Externals **9**
2.1 Supported Constructs . . . 9
2.2 Minimal Runtime . . . 10
2.3 Using GNAT runtime files . . . 11
2.4 Composite Assignment and Comparison . . . 12

3 Ada Features Useful for Generating C Code **15**
3.1 Relevant Pragmas . . . 15
3.2 Dynamic Memory Handling . . . 15
3.3 Enabling/Disabling Runtime Checks . . . 16
3.4 Enabling/Disabling Runtime Assertions . . . 16
3.5 Exception Handling . . . 16
3.6 Inserting C Fragments . . . 17

4 Defining your Target Environment **19**
4.1 Compiler Structure . . . 19
4.2 Basic Switches . . . 20
4.3 Target Configuration . . . 21
4.4 LLVM Data Layout String . . . 22
4.5 Target Configuration File . . . 23
4.6 Describing the Target Compiler . . . 23
4.7 Target Compiler Parameters . . . 25

5 Using and Understanding the Generated C Code **27**
5.1 Output File Organization and Naming . . . 27
5.2 Calling Ada Code from C code . . . 28
5.3 Debugging . . . 28
5.4 Simple Code Generation Example . . . 29
5.5 Complex Code Generation Example . . . 32
5.6 Type and Object Layout . . . 35
5.7 Records, Bitfields, and Packing . . . 37

A GNU Free Documentation License **41**

This page is intentionally left blank.

CHAPTER

ONE

GETTING STARTED

This chapter provides an overview of the GNAT Pro CCG product and describes how to use it to compile an application to run on your target.

1.1 Introduction

The GNAT Pro Common Code Generator (also known as "GNAT Pro CCG" and referred to in this manual as "CCG") is a compiler based on the GNAT Pro technology. You pass your source code, written in a subset of Ada, through CCG and it generates C code with the same semantics. You complete the process by compiling this code with a C compiler for your target. In a sense, C is being used as a high-level and portable "assembly language" to compile Ada source code.

When reading this this manual, you should also refer to the GNAT, GPRbuild, and GNAT Studio documentation (User's Guides and Reference Manuals).

1.2 Calling and Using CCG

To most easily use CCG, you should create a project file (`.gpr`) as described in Setting Up a Project File in the CodePeer User's Guide and GNAT Project Manager in the GPRbuild User's Guide,.

You need to at least identify in that project file the directories where you placed your Ada source files. You should, but aren't required to, also specify an object directory, which is where CCG places the generated C code. Here's an example of a project file:

```
project My_Project is
   for Source_Dirs use (".", "src1", "src2");  -- where your source files are
   for Object_Dir use "obj";  --  where CCG places the compilation output
   for Main use ("main_unit.adb");  -- the name of the main file

   package Compiler is
      for Switches ("Ada") use
        ("-gnatp");  -- switches used for compiling Ada files
   end Compiler;
end My_Project;
```

To generate C code for your project, you run `gprbuild` and specify the special `c` target, either via the `--target` command line switch or the `Target` project file attribute:

```
$ gprbuild -p --target=c -Pmy_project
```

or:

```
project My_Project is
   for Target use "c";
   for Source_Dirs use ...;
   --  other project properties
end My_Project;
```

```
$ gprbuild -p -Pmy_project
```

`gprbuild` knows which files need to be compiled, generates a `.c` file for each Ada unit in your project, and places them in your project's object directory. In the same directory, it also creates both Ada and C versions of a binder procedure, which contains code to initialize the runtime and invoke the main unit (for a commented example of a binder file, see this example).

It's your responsibility to call your target C compiler and linker since how to do that depends on your target's environment. For example, if you're using the GCC C compiler, you can do:

```
$ cd obj
$ gcc *.c -o main
```

The above doesn't include any runtime files in your program. See *Using GNAT runtime files* (page 11) for information on how to find and include any runtime files you may need.

If you want to compile a file manually, you can use the `c-gcc` command with the same arguments you'd pass to the `gcc` command when using the native version of GNAT Pro as well as any needed CCG-specific switches.

CHAPTER

TWO

SUPPORTED CONSTRUCTS, LIBRARIES, AND NEEDED EXTERNALS

This chapter discusses which Ada features are supported by CCG, how to use them (especially those that require run-time support), and when your application needs to reference externally-provided functions.

2.1 Supported Constructs

CCG supports a subset of Ada features. Some require run-time support; see *Using GNAT runtime files* (page 11) for more details. Here's a list of the constructs currently supported:

- packages (including child, separate, and generic packages)
- subprograms (including functions returning unconstrained arrays or dynamically sized records and separate, generic, nested, and overloaded subprograms)
- all discrete types whose base type has a size of up to 64 bits (128 bits base types are not supported)
- all other Ada types and subtypes
- all representation clauses, including packed arrays
- most attributes, including 'Image on scalars
- runtime checks
- assertions, preconditions, and postconditions
- raising an exception (with an explicit `raise` or implicitly via a runtime check or an assertion failure), which is mapped to a call to a last chance handler subprogram unless caught within the same function that raised it. See *Exception Handling* (page 16) for more details.
- `delay until` statement

The following Ada constructs are not currently supported:

- Those which require significant run-time support:
  - tasking
  - controlled types
  - interface types
  - interprocedural exception handling
  - storage pools
- `'Image` on floating- and fixed-point types
- Assembly insertion

  If you need to include assembly code, you can do so by putting the assembly code in a separate file.

Other Ada features require library files that aren't currently included in the set provided by CCG, but it's possible to add more runtime files to support them; please contact AdaCore support if you need to do this.

2.2 Minimal Runtime

CCG comes with a minimal runtime that includes the following packages:

- `Ada`
- `Ada.Assertions`
- `Ada.Numerics`
- `Ada.Numerics.Elementary_Functions` and `Ada.Numerics.Long_Elementary_Functions`

  We include simplified versions of these packages that map directly to the underlying `math.h` C library.
- `Ada.Real_Time`
  - Support for `Ada.Real_Time.Clock` requires you to provide the following two functions tailored for your target:

    ```
    unsigned long rts_monotonic_clock (void);
    /*  Returns time from the board/OS  */

    int rts_rt_resolution (void);
    /*  Returns resolution of the underlying clock used to implement
        Monotonic_Clock.  */
    ```

  - Support for the `delay until` statement requires that you provide the following external function:

    ```
    void rts_delay_until (unsigned long t);
    /*  Delay until a given time as returned by rts_monotonic_clock  */
    ```

- `Ada.Text_IO` and `Text_IO`

  We include a simplified version for doing simple output of characters and strings, but not input. If you want to output other types, you can use the `'Image` attribute to transform a value into a string. This package depends on the C standard function `putchar`.
- `Ada.Unchecked_Conversion` and `Unchecked_Conversion`
- `Ada.Unchecked_Deallocation` and `Unchecked_Deallocation`
- `Interfaces`
- `Interfaces.C`
- `Interfaces.C.Extensions`
- `Interfaces.C.Strings`
- `System`
- `System.Storage_Elements`
- `GNAT`
- `GNAT.Source_Info`

2.3 Using GNAT runtime files

Some Ada constructs require support from GNAT runtime units. You need to include the files containing the units that support those constructs as part of your C build if and only if you use these Ada constructs.

These constructs include:

- functions returning dynamically-sized objects
- the `'Image` attribute
- packed array component sizes that aren't a power of 2
- exponentiation

You can locate the directory containing the runtime by executing the following command:

```
$ c-gnatls -v | grep adainclude
```

The corresponding `.c` files for these units are located in the `adalib` directory and have been generated for the default (32 bits) target configured in CCG.

If this default target is suitable, you can copy the needed C files to your C build directory and compile these files with your target C compiler in the same way you compile the C files generated by CCG. Your linker should tell you about missing symbols, so successfully linking your Ada application means you have included all needed GNAT runtime files.

If you are using a non-default target configuration, you must first recompile each needed file.

If you you know exactly which runtime files you need and it's only a few of them, you can use the following command on each of these files individually (note the use of the `-gnatpg` switch):

```
$ c-gcc -c <any relevant switches> -gnatpg </path/to/runtime/file>
```

This creates a new `.c` file in the current directory. You can use it instead of the pre-built version of that file. Relevant switches may include `-gnateT` and `--target`, depending on your requirements, and should be the same as those you use to compile the rest of your application.

If you don't know which runtime files you need, or if you need more than a few, you should instead first copy the contents of the `adainclude` directory into a local directory in your project, e.g:

```
$ cp -pr `c-gnatls -v | grep adainclude` .
```

Then create a project file named `rts.gpr` that you will use to compile this local copy of adainclude:

```
project RTS is
   for Source_Dirs use ("adainclude");
   for Object_Dir use "obj";  --  you can use the same object directory
                              --  as your main project

   package Compiler is
      for Switches ("Ada") use ("-gnatpg");
      --  Add other relevant switches such as --target=
   end Compiler;
end RTS;
```

Finally, add a `with "rts"` in your main project file, e.g:

```
with "rts";
project Main is
  ...
end Main;
```

When you've done this, building your main project will also compile the runtime files you need. E.g.:

```
$ gprbuild -Pmain --target=c
```

The above command will compile your project, including all the runtime files it uses, and generate the corresponding C files in the object dir specified in `rts.gpr`.

You need to take an additional step if your code is using the secondary stack. You need to explicitly compile the file `s-sssita.adb`:

```
$ gprbuild -q -c -u -Pmain s-sssita.adb --target=c
```

See also *Composite Assignment and Comparison* (page 12) for other external dependencies generated by CCG.

2.4 Composite Assignment and Comparison

To support composite assignments (records and arrays), CCG may generate calls to the standard C routines `memcpy` and `memmove`. Similarly, CCG may generate calls to `memcmp` for composite comparisons.

If your target C compiler doesn't provide these routines, you have several options:

- remove the composite assignments or comparisons from your code, for example by replacing array assignments with loops over each element
- provide your own implementation of these functions

For example, here's a simple implementation of `memcpy` that you can add as part of your project, assuming you placed `#include <stddef.h>` earlier in your file:

```
void *
memcpy (void *dest, const void *src, size_t n)
{
  char *src_p = (char *) src;
  char *dest_p = (char *) dest;
  size_t i;

  if (n == 0)
    return dest;

  /* Copy contents of src[] to dest[] backwards. Coding the loop this way
     properly handles the case of n == SIZE_MAX. */

  i = n - 1;
  do {
    dest_p[i] = src_p[i];
  } while (i-- > 0);

  return dest;
}
```

Similarly for `memmove`:

```
void *
memmove (void *dest, const void *src, size_t n)
{
  char *src_p = (char *) src;
  char *dest_p = (char *) dest;
  size_t i;

  if (n == 0)
```

(continues on next page)

(continued from previous page)

```
    return dest;

  /* This function must handle overlapping memory regions
     for the source and destination. If the destination buffer
     is located in the middle of the source buffer, we copy
     backwards. Otherwise, we copy in the forward direction.     */

  if (dest > src && dest < src + n)
    {
      i = n - 1;
      do {
        dest_p[i] = src_p[i];
      } while (i-- > 0);
    }
  else
    {
      /* Copy in two parts to properly handle the case of n == SIZE_MAX */
      for (i = 0; i < n - 1; i++)
        dest_p[i] = src_p[i];

      /* i == n - 1 at this point */
      dest_p[i] = src_p[i];
    }

  return dest;
}
```

And memcmp:

```
int
memcmp (const void *s1, const void *s2, size_t n)
{
  char *s1_p = (char *) s1;
  char *s2_p = (char *) s2;
  size_t i;

  if (n == 0)
    return 0;

  for (i = 0; i < n - 1; i++)
    {
      if (s1_p[i] < s2_p[i])
        return -1;
      else if (s1_p[i] > s2_p[i])
        return 1;
    }

  /* i == n - 1 at this point */
  if (s1_p[i] < s2_p[i])
    return -1;
  else if (s1_p[i] > s2_p[i])
    return 1;
  else
    return 0;
}
```

This page is intentionally left blank.

CHAPTER

THREE

ADA FEATURES USEFUL FOR GENERATING C CODE

There are some features of the Ada language that are especially useful in the context of CCG. We discuss them in this chapter.

3.1 Relevant Pragmas

CCG supports most Ada and GNAT pragmas. The following are particularly relevant for CCG. You can find more details about them in the GNAT Reference Manual.

- `pragma Restrictions (No_Dynamic_Sized_Objects)`

  Ensure that code doesn't contain objects of dynamic size, which require the use of `alloca` or dynamic sized C objects, since these aren't supported by all C compilers. You can't use this pragma when your sources have tagged types since dispatch tables require dynamically-sized record types.

- `pragma Restrictions (No_Multiple_Elaboration)`

  When this restriction is active, CCG is allowed to suppress the elaboration variable normally associated with the unit, even if the unit has elaboration code. This variable is typically used to check for access before elaboration and control multiple elaboration attempts. You may want to use this pragma to reduce the size of the generated code and lower the number of global symbols in C files.

- `pragma Discard_Names`

  Removes the generation of internal strings, in particular for enumeration types, generating simpler and smaller C code. This pragma also disables the support for `'Image` on enumeration types.

- `pragma Suppress_Exception_Locations`

  Suppress messages associated with exceptions (in particular assertions, preconditions, postconditions, and predicates) to reduce the memory footprint.

3.2 Dynamic Memory Handling

CCG supports the use of dynamic memory by generating calls to the standard C functions `malloc` (for allocation) and `free` (for deallocation). If you use dynamic memory in your Ada sources, your C compiler's library or your own code must provide one or both of those functions.

3.3 Enabling/Disabling Runtime Checks

By default, CCG enables all runtime checks except overflow checks. If you want to suppress the generation of checks (e.g. because they have been proven by the SPARK toolset or for efficiency purposes), you can do one of the following:

- use the `-gnatp` compiler switch, which disables all runtime checks
- place the follwing in your configuration file (named `config.adc` by default):

```
pragma Suppress (All_Checks);
```

  To enable the use of `config.adc` as your configuration file, you need to add the following to your project file:

```
for Global_Configuration_Pragmas use "config.adc";
```

- selectively use `pragma Suppress` in a source or configuration file to suppress some checks or to suppress checks only for certain packages or subprograms, e.g.:

```
pragma Suppress (Range_Checks);  -- suppress range checks only
```

  or in a source file:

```
procedure Proc1 is
   pragma Suppress (All_Checks);  -- suppress all checks for this procedure
begin
   -- ...
```

See the GNAT documentation for more details.

3.4 Enabling/Disabling Runtime Assertions

By default, CCG doesn't generate code for assertions (including pre- and postconditions). You have several ways to enable assertions in the generated code:

- use the `-gnata` compiler switch to enable all assertions.
- use `pragma Assertion_Policy` to selectively enable or disable assertions.

See the GNAT documentation for more details.

3.5 Exception Handling

If you enable any runtime checks or assertions or if you have explicit `raise` statements in your source and these aren't caught by exception handlers in the same function in which they're raised, CCG generates calls (conditional in case of a runtime check or assertion) to a subprogram with the following C profile:

```
extern void __gnat_last_chance_handler (const char *file, int line);
```

You must provide this function as part of your code in those circumstances. The parameter `file`, if not `NULL`, is a C string (null-terminated) with the name of the Ada file where the exception was raised. The second parameter corresponds to the line number in that file. This function can perform some activities (e.g., logging) and is responsible for stopping or restarting the application. If this function returns normally, further execution is undefined.

3.6 Inserting C Fragments

You may find it useful to insert compiler-specific code fragments into the generated C code.

For example, to include `#pragma` directives, you can use `pragma Annotate` as follows:

```
pragma Annotate (CCG, C_Pragma, "insert pragma contents");
```

which inserts:

```
#pragma insert pragma contents
```

in the C code starting on a new line at column 1.

To generate an `#include` directive, you can use `pragma Annotate` as follows:

```
pragma Annotate (CCG, Include, "filename.h");
```

which inserts:

```
#include "filename.h"
```

in the C code starting on a new line at column 1.

You can also use a more general syntax to insert arbitrary code:

```
pragma Annotate (CCG, Verbatim, "any valid C code");
```

which will insert whatever code you specified on a new line starting in column one.

You should use this capability with great care and manually review the resulting output for correctness. CCG makes no attempt to verify the contents of the C code provided, so improper use of this pragma may generate invalid or incorrect C code.

CCG tries to generate C code for objects and subprograms in the same order as in the original Ada source and hence tries to place these insertions at locations in the C source that correspond to their placement in the Ada source, but this isn't always possible and CCG doesn't guarantee that the location of the inserted C code is what you expect. You should always manually review the placement of each to verify it was placed where you expected.

Because of how CCG processes types, using these fragments to change the way that types are handled probably won't produce the desired effects. See the following chapter for ways to affect the processing of types.

If you manually include one or more `.h` files in your C output, you can indicate that an external function or global you wish to use is to defined in one of those files and that CCG is not to produce a declaration for it by prefixing '#' in front of the external name of the function. For example:

```
function F return Integer
  with Import, Convention => C, External_Name => "#f";
X : Integer with Import, External_Name => "#x";
```

When you do this, CCG refers to function `F` using an external name of `f`, but won't generate a declaration for this function. Likewise, it will refer to the global `X` using an external name of `x`, but won't generate a declaration for the global. It's your responsibility to provide proper definition of any such functions and globals in a file you specify in a `pragma Annotate` directive.

If you aren't certain what that definition should look like, you may want to first omit the '#', see what the definition is, and then copy that into your `include` file.

This page is intentionally left blank.

CHAPTER

FOUR

DEFINING YOUR TARGET ENVIRONMENT

Most users of CCG are generating code for a very specific environment, consisting of a target CPU, a target C compiler, and other supporting tools and libraries. To make the most effective use of CCG, you need to describe your environment. This chapter tells you how to do that.

4.1 Compiler Structure

CCG uses the GNAT Pro front end and the LLVM backend to generate LLVM IR (Internal Representation) and then translates the LLVM IR into C. The LLVM IR is a portable assembly language with an infinite number of named registers and in SSA (Static Single Assignment) format. Each switch you specify affects the way the front end processes Ada code, the way Ada code is translated into LLVM IR, or the way the generated LLVM IR is translated to C.

Each object in the LLVM IR has a type. The most common `first class` types are:

- `void`
- `i<n>`

  an `n` bit integer type
- `float`

  a 32-bit floating-point value
- `double`

  a 64-bit floating-point value

Composite types include pointers, arrays, records, and functions.

For example, consider the following tiny Ada function:

```
function Add (X, Y : Integer) return Integer is
begin
   return X + Y;
end Add;
```

This generates the following LLVM IR (not including debug information):

```
; ModuleID = 'add.adb'
source_filename = "add.adb"
target datalayout = "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-f64:32:64-f80:32-
↪n8:16:32-S128"
target triple = "i386-linux"

define i32 @_ada_add(i32 %x, i32 %y) {
entry:
  %0 = add nsw i32 %x, %y
```

(continues on next page)

(continued from previous page)

```
  ret i32 %0
}
```

4.2 Basic Switches

Here are some switches you can use to influence the generation of C code:

- `-g`

  Enable debug information. In this context, debug information means generating *#line* directives; see *Debugging* (page 28).

- `-gnatL`

  Include the original Ada code interspersed as C comments in the generated code. You can use this option to help debug the C code; see *Debugging* (page 28).

- `-gnatp`

  Suppress all runtime checks; see *Enabling/Disabling Runtime Checks* (page 16).

- `-gnata`

  Enable all assertions; see *Enabling/Disabling Runtime Checks* (page 16).

- `-O[123s]`

  Generate optimized C code. By default, CCG performs no optimization when generating C code (`-O0`). `-O1` enables many optimizations, `-O2` enables even more (and more computation intensive) optimizations, and `-O3` enables maximum optimization. `-Os` is similar to `-O2` and will favor code size when possible. These optimizations don't include target-specific optimizations, so you should also pass an optimization switch to your C compiler if you want the highest code performance.

- `-fuse-gnat-allocs`

  GNAT LLVM generates calls to `__gnat_malloc` and `__gnat_free` for allocation and deallocation, respectively. By default, CCG generates calls to `malloc` and `free`. When you specify this switch, CCG generates C code calling the GNAT versions of these functions.

- `-fuse-stdint`

  Use C types defined in `<stdint.h>` instead of the predefined C types for integers. See *Output File Organization and Naming* (page 27) for an example of how this changes the output. Note that CCG will still use the type name in `Interfaces.C` for Ada types in or derived from those in that unit even if you specify this switch.

- `-fprefer-packed`

  Most records are internally represented as packed. By default, CCG tries to write each record's C equivalent as a non-packed record when possible. Specify this switch if you want to suppress this rewriting and have CCG keep most records packed. This switch is illegal if you indicate your target C compiler doesn't support packing.

- `-emit-header`

  Emit a header file (`<source basename>.h`) instead of generating a C file. You can include this file, which contains only type, variable, and function declarations, in manually-written C code. CCG also includes C declarations of public Ada enumeration types into this file even though those enumeration type names won't appear in the generated C code. These files aren't needed or referenced by code generated by CCG. See *Calling Ada Code from C code* (page 28) for more information on using these files.

- `-header-inline={none,inline,inline-always}`

  When you specify `-emit-header`, you use this switch to tell CCG which, if any, bodies of inlined functions it should write to the `.h` file. The default is `none`.

4.3 Target Configuration

You can specify attributes of both your target CPU and target C compiler. The next few sections discuss parameters for your target CPU.

When using CCG, the code for your target is generated by your target C compiler and it's responsible for data and memory layout and alignment. However, there are many case where CCG needs to know, in detail, what the C compiler you're using will do in those areas. It needs this information, for example, to know what size objects are and when it needs to take actions to override the C compiler's default (such as needing to pack a struct or add an alignment attribute). This section describes how you tell CCG about what your C compiler will do. You need to do this to ensure that CCG generates correct code. But keep in mind that this information is only used for that specific purpose: the actual storage layout is determined by your target C compiler.

By default, CCG assumes a 32-bit, little-endian target. To describe your actual target, you may need to specify parameters that provide details about your target, including its endianness and the sizes and alignments of pointer and integer types. The starting point for all target parameterization is what's known as a "target triple", whose full form is three values separated by dashes. Usually only two are specified: the architecture and the operating system. For example, `x86-linux`.

A CCG binary is built with a subset of the targets supported by LLVM. To display both the current triple and the list of included targets, include the `--dump-targets` switch when compiling a file. Each target corresponds to a set of supported triples that start with a name related to that target. Currently, CCG includes support for the 32-bit `x86` and 64-bit `x86-64` targets (triples starting with `i386` and `x86_64`, respectively), as well as AArch64 targets (triples starting with `aarch64`, `aarch64_32`, and `aarch64_be`). For all targets, the supported operating systems include `linux`, `elf`, and `windows`.

You should start with the target supported by the CCG binary that's closest to your target. For example, for a 64-bit target that doesn't use an operating system, you'd start with a target triple of `x86_64-elf` and for a 32-bit embedded Linux system, you'd start with `i386-linux`. Then further modify target parameters, if needed, by providing an LLVM data layout string (see *LLVM Data Layout String* (page 22)), a target configuration file (see *Target Configuration File* (page 23)), or both.

Because CCG is generating C code and not machine code, there's no requirement that your CPU be that given by the target triple (and indeed it often won't be), just that you ensure that all the relevant target parameters (data sizes and layout information) used by CCG correspond to those of your target CPU. You provide these parameters using the following switches:

- `-mtriple=`
- `--target=`

  The supported LLVM triple closest to your target. For `--target`, you can specify both a target triple and an LLVM data layout string, separated by a colon, but this use is deprecated in favor of the `--layout` switch below.

- `--layout=`

  A LLVM data layout string for your target, if needed.

- `-gnateT=`

  A target configuration file for your target, if needed.

4.4 LLVM Data Layout String

The LLVM Data Layout String describes how data is laid out in memory. It consists of a list of specifications separated by minus signs. Each specification starts with a letter and may include other information to define some aspect of the data layout. You can find documentation of the full syntax of the LLVM data layout string in the LLVM Language Reference Manual.

Unlike the target configuration file discussed in *Target Configuration File* (page 23), this string only describes the way the target lays out and aligns objects of various sizes, not the sizes used for various types (such as `int`). For the most precise description of a target, you may need to specify both an LLVM Data Layout String and a target configuration file.

As stated in its documentation, LLVM constructs the data layout for a given target by starting with a default set of specifications which are then overridden by any specifications in the data layout string.

The most relevant default specifications are below:

- e

  little endian

- p:64:64:64

  64-bit pointers with 64-bit alignment.

- S0

  natural stack alignment is unspecified

- i1:8:8

  i1 is 8-bit (byte) aligned

- i8:8:8

  i8 is 8-bit (byte) aligned

- i16:16:16

  i16 is 16-bit aligned

- i32:32:32

  i32 is 32-bit aligned

- i64:32:64

  i64 has ABI alignment of 32-bits but preferred alignment of 64-bits

- f32:32:32

  float is 32-bit aligned

- f64:64:64

  double is 64-bit aligned

- a:0:64

  aggregates are 64-bit aligned

The relevant portion of two sample LLVM data layout strings are:

- `e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128`.

  a data layout string corresponding to the LLVM target triple `i386-linux`

- `E-n32-A0-p32:32f64:0:0-i64:0:0`

  a data layout string for a PowerPC 750 processor configured as big-endian

4.5 Target Configuration File

If you choose to specify a target configuration file, you can reference it in your project file as follows:

```
package Builder is
   for Global_Compilation_Switches ("Ada") use
     ("-gnateT=" & project'Project_Dir & "/target.atp");
end Builder;
```

where `target.atp` is a file you've placed in the same directory as your project file and which contains your target's parameterization. The format of this file is described in the GNAT User's Guide as part of the description of the `-gnateT` switch.

Note that CCG only supports integer types up to 64 bits, so you can't specify a value of 128 for any of these parameters.

Here's an example of a configuration file for a bare board PowerPC 750 processor configured as big-endian:

```
Bits_BE                       1
Bits_Per_Unit                 8
Bits_Per_Word                32
Bytes_BE                      1
Char_Size                     8
Double_Float_Alignment        0
Double_Scalar_Alignment       0
Double_Size                  64
Float_Size                   32
Float_Words_BE                1
Int_Size                     32
Long_Double_Size             64
Long_Long_Size               64
Long_Long_Long_Size          64
Long_Size                    32
Maximum_Alignment            16
Max_Unaligned_Field          64
Pointer_Size                 32
Short_Enums                   0
Short_Size                   16
Strict_Alignment              1
System_Allocator_Alignment    8
Wchar_T_Size                 32
Words_BE                      1

float          6  I  32  32
double        15  I  64  64
long double   15  I  64  64
```

4.6 Describing the Target Compiler

To allow CCG to generate the C code most appropriate for your environment, you may need to specify some information about the C compiler you're using to compile the generated code. You can specify one of a list of C compilers closest to your compiler as a starting point and, if necessary, provide options to further tune the code to that required by your target compiler. This is important in the case of extensions to C, such as those required to specify that records are packed or that objects require nondefault alignment or need to be placed by the linker in nondefault sections.

Here are the switches you can use to provide that information:

- `-c-compiler=<compiler-name>`

  Tells CCG the compiler family you're using. It uses this to set initial values of the target compiler parameters. CCG currently supports the values `gcc`, `clang`, `msvc`, and `generic`. The default is `gcc`. Use the `generic` option for a C compiler that doesn't support any extensions to the C language.

- `-c-target-<parameter-name>=<value>`

  Sets the value of target parameter `parameter-name` to the specified value. See *Target Compiler Parameters* (page 25) for a list of parameters and their possible values.

- `c-target-modifier-<modifier-name>=<value>`

  Sets the value of the modifier `modifier-name` to the specified value. A *modifier* is a string that's added to a declaration to specify some attribute, such as packing or alignment. Different compilers, such as GCC and MSVC, use different syntax for such modifiers. You specify the string to use for a modifer as `value`. If that modifier has a parameter, such as an alignment or section name, indicate the place to put that value with a percent character. If your target compiler doesn't support a modifier, specify a string of just a dollar sign (but see the description of `packed-mechanism` below for a special case involving packing). The default value for a modifier (which is not output in the target parameter file) is the syntax used for GCC. For example, the default `section` modifier is `-c-target-modifier-section=__attribute__((section(%))`.

  You can specify the names to be used for custom modifiers by using the target parameters discussed in *Target Compiler Parameters* (page 25). The following modifiers are built in to CCG:

  - `aligned`

    Indicates that this declaration requires a nondefault alignment; the alignment is the parameter for this modifier.

  - `always_inline`

    Indicates that the subprogram being declared is to always be inlined.

  - `noreturn`

    Indicates that the subprogram being declared will never return.

  - `packed`

    Indicates that the record being declared needs to be packed; see *Records, Bitfields, and Packing* (page 37).

  - `section`

    Specifies the section into which the linker is to place the object being declared.

  - `stdcall`

    Specifies how the compiler marks functions using StdCall calling convention.

- `-c-target-file=<filename>`

  Reads compiler target parameters and modifiers from `filename`. You write this file with one parameter or modifier per line, each of the form `<parameter-name>=value` or `modifier-<modifier-name>=value`.

- `--dump-c-parameters=<filename>`

  Creates `filename` containing the current values (after processing all previous switches) of the compiler target parameters and modifiers. This file is in the format read by the `-c-target-file` switch.

With the default values, the compiler target file looks like this:

```
version=1999
indent=2
max-depth=10
always-brace=False
parens=warns
have-includes=True
```

(continues on next page)

(continued from previous page)

```
inline-always-must=True
inline-style=std
code-section-modifier=section
declare-section-modifier=$
packed-mechanism=modifier
```

When specifying `-c-compiler=msvc`, it looks like this:

```
version=1999
indent=2
max-depth=10
always-brace=False
parens=warns
have-includes=True
inline-always-must=True
inline-style=std
code-section-modifier=code-seg
declare-section-modifier=decl_sect
packed-mechanism=pragma
modifier-section=__declspec(allocate(%))
modifier-code-seg=__declspec(code_seg(%))
modifier-decl_sect=#pragma section(%)
modifier-always_inline=$
modifier-noreturn=__declspec(noreturn)
modifier-aligned=__declspec(align(%))
modifier-stdcall=__stdcall;
```

4.7 Target Compiler Parameters

Here is a list of the parameters you can use to tune the generated C code for your target C compiler:

- `version`

  The version of the C standard that CCG is generating code for. The default is C99. CCG uses this to determine how to generate C corresponding to some Ada features. For example, we use empty brackets for C99 or later as the dimension of an array to show that it's of variable length. For C versions older the C90, we don't support `alloca`.

  You can specify this parameter in a number of ways. For example, for C90 you can write `90`, `1990`, or `C90`.

- `indent`

  The number of characters to indent the generated code by at each level of nesting. The default is 2.

- `max-depth`

  The maximum allowable nesting depth of constructs. If CCG needs to generate nesting beyond this point, it generates a branch to out-of-line code. The default is 10.

- `always-brace`

  True if CCG is to always write C lexical blocks using braces even if they're only a single line. The default is `False`.

- `have-includes`

  True if CCG is to write `#include` lines for the standard C include files. The default is True.

- `parens`

  Tells CCG when to output parentheses. You can select one of the following options:

  - `always`

    Output parentheses around each subexpression. Specify this if you suspect problems due to precedence issues or if you prefer reading C code with the precedence explicit.

  - `normal`

    Only output parentheses when they aren't implied by the precedence rules.

  - `warns`

    Output parentheses when they aren't implied by the precedence rules or if your C compiler will issue warning in cases where the precedence is correct but looks suspicious. This is the default.

- `inline-always-must`

  In some C compilers, such as `clang`, `Inline_Always` means to make a best try at inlining, but be silent if the function can't be inlined. In others, such as `gcc`, if the function can't be inlined, it issues a warning (or error, depending on the warning mode). The value of this option says which is the case. The default is `True`.

- `inline-style`

  The standard way of requesting a C function be inlined is to use the keyword `inline`. MSVC supports `__inline` and some older versions only support that form. Specify the value `std` for this parameter to use the standard form and `msvc` for the MSVC form. The default is `std`. (Because most versions of MSVC support the standard form, specifying `-c-compiler=msvc` does not change this parameter.)

- `code-section-modifier`

  In some compilers, such as MSVC, you indicate that the linker is to put a function in a nondefault section in a different manner than you would indicate a nondefault section for data. For such a compiler, use this parameter to supply the name of a modifier to use to indicate the code section. You will usually need to specify a syntax for that modifier using the `-c-target-<modifier>=<value>` switch. For example, for MSVC, you could have `-c-target-declare-section-modifier=decl_sect` and `c-target-modifier-code-seg=__declspec(code_seg(%))`. The default is `section`.

- `declare-section-modifier`

  In some compilers, again such as MSVC, if you specify a linker section name for data (but not functions), that section must be separately declared. For such a compiler, use this modifier to specify the syntax to declare that section. For example, you could specify `-c-target-declare-section-modifier=decl_sect` and `-c-target-modifier-decl_sect="#pragma section(%)"` for MSVC. If such a declaration isn't needed, specify a dollar sign as the name of the modifier (this is the default).

- `packed-mechanism`

  CCG currently supports compilers that use one of two methods to indicate that a record is packed. You specify which, if any, of those methods is used by your compiler by setting this parameter to one of the following values:

  - `modifier`

    Specify this value if your compiler uses a modifier (`packed`) to declare a record as packed.

  - `pragma`

    Specify this value if your compiler uses pragmas (in the MSVC syntax) to indicate that a record is packed.

  - `none`

    Specify this value if your compiler supports neither of those ways to denote that a record must be packed. In this case, CCG generates an error if your program requires packing in order to be correctly compiled. If your compiler supports a different mechanism than the above and you want to use packed records, please contact AdaCore support.

CHAPTER

FIVE

USING AND UNDERSTANDING THE GENERATED C CODE

In most cases, you can view the C produced by CCG in much the same manner as you'd view assembly code in a native environment: something you don't need to look at or understand. But there are times, especially when debugging or if you have a mixed C and Ada environment, where you need to be able to understand the C code produced by CCG and relate it to your Ada program. This chapter contains information to help you in those situations, as well as tips to help you debug your program when using CCG.

5.1 Output File Organization and Naming

As discussed above, CCG generates one C file (named `<source basename>.c`) for each compiled Ada source file. Each file contains all the statements and declarations for the unit spec and body, including declarations for any external data and subprograms used by the spec and body.

You can request CCG to instead generate a header file for a compiled Ada source file by specifying the `-emit-header` switch (see *Basic Switches* (page 20)). This causes CCG to generate a separate `<source basename>.h` file, intended for use in manually-written C code, that includes all the type, subprogram, and data declarations. You can specify the `-header-inline` switch to say which, if any, inlined functions CCG should also include in that file.

CCG encodes each global entity (variable, constant, subprogram) into a name that's fully documented in GNAT source file `exp_dbug.ads`. In general, entities in package `Pack1` are translated into C symbols named `pack1__<entity>` (all lowercase). For example, subprogram `Pack1.Subprogram1` becomes a C function called `pack1__subprogram1`.

Here's an example in Ada, showing the translation into C with the default switches:

```
pragma Restrictions (No_Multiple_Elaboration);
package Pack1 is
   type Enum is (A, B, C) with Discard_Names;

   X : Enum;
   Y : Float;
end Pack1;
```

```
#include <string.h>
#include <stdlib.h>

unsigned char pack1__x = 0;
float pack1__y = 0.0e+00f;
```

If you specify the `-fuse-stdint` switch, the resulting file will look like this:

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
uint8_t pack1__x = 0;
float pack1__y = 0.0e+00f;
```

5.2 Calling Ada Code from C code

To call Ada subprograms and use corresponding data structures from C code, you can `#include` any needed `.h` files that CCG generated when you specified the `-emit-header` switch. Using the example from the previous section, CCG produced a file `pack1.h` containing:

```
#ifndef PACK1_ADS_H
#define PACK1_ADS_H

enum  {
    pack1__enum__a = 0,
    pack1__enum__b = 1,
    pack1__enum__c = 2
};

typedef int8_t pack1__enum;


extern unsigned char pack1__x;
extern float pack1__y;

#endif /* PACK1_ADS_H*/
```

If you want to include these definitions for `pack1.ads` in any of your C files, you need to put the following line in each of those files:

```
#include "pack1.h"
```

Note that the first two and the last line of `pack1.h` allow you to include that file multiple times.

Also note that `pack1.h` doesn't supply a tag for the C enum corresponding to the Ada type `Enum`, but instead provides a `typedef`. This is because C compilers use an `int` or wider integral type for enums, but Ada, by default, uses the smallest integer type that holds the enum values.

5.3 Debugging

You have several options for debugging the generated code, each with some associated limitations:

- host debugging using a native Ada compiler

  This is a possibility if you can run some or all of your application on a system supported by GNAT Pro. This option provides a powerful debugging experience, but doesn't allow debugging target-specific issues.

- debugging the generated C code using a C debugger

  This option requires no specific toolchain support and allows debugging target-specific issues, but requires you to become familiar with the generated C code. Using the `-gnatL` switch can help debugging the C code by having the Ada code also available as comments; see *Basic Switches* (page 20). If you specify an optimization switch other than `-O0`, it may be difficult to relate the C code to the Ada code even when you also specify `-gnatL`.

- debugging the Ada code on your target using a C debugger

You can enable this hybrid solution by including the `-g` compiler switch to CCG, including the corresponding debug switch for your target's C compiler, and using your target's C debugger. When you specify this switch, CCG inserts `#line` directives in the generated C code, so that messages (errors or warnings) from the C compiler are redirected to the Ada source code and debug information points to the Ada code, allowing you to perform step-by-step debugging at the Ada source level. This capability depends on the ability of your target C compiler to properly handle `#line` directives pointing to external non-C files. You also need some knowledge of the generated C code to display values of Ada variables properly, in particular to use the proper encoding for global variables and subprograms.

5.4 Simple Code Generation Example

Let's look at the following package, which uses built-in features of Ada, in this case exponentiation and a function returning an array:

```
package P is

   type Int is private;
   type Int_Array is array (1 .. 10) of Int;

   function Square (X : Int) return Int;
   function Square (X : Int_Array) return Int_Array;

private
   type Int is new Integer;
end P;
```

```
package body P is

   function Square (X : Int) return Int is
   begin
      return X ** 2;
   end Square;

   function Square (X : Int_Array) return Int_Array is
      Result : Int_Array;
   begin
      for J in X'Range loop
         Result (J) := X (J) ** 2;
      end loop;

      return Result;
   end Square;

end P;
```

CCG generates the following C code when compiled with `c-gcc -c -gnatp -gnatL p.adb`:

```
#include <string.h>
#include <stdlib.h>

/* 1: package body P is */
typedef int ccg_a1[10];

short p_E = 0;
int p__square (int);
void p__square__2 (ccg_a1 *, ccg_a1 *);
```

(continues on next page)

(continued from previous page)

```
 /* 3:    function Square (X : Int) return Int is */
 int p__square (int x)
 {
 /* 4:    begin */
 /* 5:       return X ** 2; */
   return x * x;
 }

 /* 6:    end Square; */
 /* 8:    function Square (X : Int_Array) return Int_Array is */

 void p__square__2 (ccg_a1 * _return_, ccg_a1 * x)
 {
 /* 9:       Result : Int_Array; */
   ccg_a1 result;
/* 10:    begin */
/* 11:       for J in X'Range loop */
   int j;
   int ccg_v2;

   j = 1;

/* 12:          Result (J) := X (J) ** 2; */
ccg_l3:
   result[j - 1] = (*x)[j - 1] * (*x)[j - 1];
   ccg_v2 = j;
   j = ccg_v2 + 1;
   if (ccg_v2 == 10)
 /* 13:       end loop; */
 /* 15:       return Result; */
     {
       memcpy ((char *) _return_, (char *) &result, 40);
       return;
     }

   goto ccg_l3;
 }
/* 16:    end Square; */
/* 18: end P; */
```

We've used the `-gnatL` switch to include the Ada source before the C code generated from each line of Ada.

And if you enable the optimizer via `c-gcc -c -O -gnatp -gnatL p.adb`, CCG generates the following:

```
#include <string.h>
#include <stdlib.h>

/* 1: package body P is */
typedef int ccg_a1[10];

short p_E = 0;
int p__square (int);
void p__square__2 (ccg_a1 *, ccg_a1 *);

/* 3:    function Square (X : Int) return Int is */
int p__square (int x)
```

(continues on next page)

(continued from previous page)

```
{
/* 4:    begin */
/* 5:       return X ** 2; */
  return x * x;
}

/* 6:    end Square; */
/* 8:    function Square (X : Int_Array) return Int_Array is */

void p__square__2 (ccg_a1 * _return_, ccg_a1 * x)
{
/* 9:       Result : Int_Array; */
/* 10:    begin */
/* 11:       for J in X'Range loop */
/* 12:          Result (J) := X (J) ** 2; */
  int ccg_v2;
  int ccg_v3;
  int ccg_v4;
  int ccg_v5;
  int ccg_v6;
  int ccg_v7;
  int ccg_v8;
  int ccg_v9;
  int ccg_v10;
  int ccg_v11;

  ccg_v2 = (*x)[0];
  ccg_v3 = (*x)[1];
  ccg_v4 = (*x)[2];
  ccg_v5 = (*x)[3];
  ccg_v6 = (*x)[4];
  ccg_v7 = (*x)[5];
  ccg_v8 = (*x)[6];
  ccg_v9 = (*x)[7];
  ccg_v10 = (*x)[8];
  ccg_v11 = (*x)[9];
/* 13:       end loop; */
/* 15:       return Result; */
  (*_return_)[0] = ccg_v2 * ccg_v2;
  (*_return_)[1] = ccg_v3 * ccg_v3;
  (*_return_)[2] = ccg_v4 * ccg_v4;
  (*_return_)[3] = ccg_v5 * ccg_v5;
  (*_return_)[4] = ccg_v6 * ccg_v6;
  (*_return_)[5] = ccg_v7 * ccg_v7;
  (*_return_)[6] = ccg_v8 * ccg_v8;
  (*_return_)[7] = ccg_v9 * ccg_v9;
  (*_return_)[8] = ccg_v10 * ccg_v10;
  (*_return_)[9] = ccg_v11 * ccg_v11;
  return;
}
/* 16:    end Square; */
/* 18: end P; */
```

To better understand this example, note the following:

- CCG originally translates Ada into LLVM IR, as described above. This IR, like assembly language, has gotos and labels. CCG attempts to create a nested set of `if`/`then`/`else` `if`/`else` blocks from this IR that

reconstructs a hierarchical view of each subprogram. This set is semantically equivalent to the original code, but doesn't always directly correspond to the way the subprogram was written in Ada.

- This version of CCG represents loops as explicit branches, labels, and tests.
- When practical, CCG uses the same name for a local variable in the C output as you used in your Ada code.
- CCG uses names of the form `ccg_v<n>` for any needed temporary variable (e.g., subexpressions used more than once).
- Optimized code may be very different from unoptimized code due to such optimizations as loop unrolling (shown here) and it may be much harder for CCG to relate expressions to named variables in your program.

5.5 Complex Code Generation Example

Now let's look at a situation where CCG is translating a construct for which there's no equivalent in C, in this case a reference to a discriminated record. This is the Ada code:

```
package Discrec is

   subtype Our_Length is Integer range 1..20;
   type R (Length : Our_Length) is record
      S1 : String (1 .. Length);
      S2 : String (1 .. Length);
   end record;

   type Ptr is access all R;
   function First_Char (P : Ptr) return Character is (P.S2 (P.S2'First));

   Rr: R (10);

end Discrec;
```

When run as `c-gcc -c -gnatp discrec.ads`, CCG produces the following:

```
#include <string.h>
#include <stdlib.h>

typedef struct discrec__r discrec__r;
typedef struct discrec__TrrS discrec__TrrS;
typedef char ccg_a1[10];
struct discrec__TrrS
{
  int length;
  ccg_a1 s1;
  ccg_a1 s2;
} __attribute__ ((packed));

typedef struct discrec__r_I discrec__r_I;
struct discrec__r_I
{
  int length;
} __attribute__ ((packed));

typedef char ccg_a2[];

void discrec__rIP (discrec__r *, int);
char discrec__first_char (discrec__r *);
```

(continues on next page)

(continued from previous page)

```
void discrec___elabs (void);
short discrec_E = 0;

inline void discrec__rIP (discrec__r * _init, int length)
{
  ((discrec__r_I *) _init)->length = length;
  return;
}

discrec__TrrS discrec__rr;

inline char discrec__first_char (discrec__r * p)
{
  return *((char *) (ccg_a2 *) &((char *) p)[((unsigned int) ((32 + 8 * ((discrec__r_
↪I *) p)->length) + 7) / 8U)]);
}

void discrec___elabs (void)
{
  discrec__rIP ((discrec__r *) &discrec__rr, 10);
  return;
}
```

When run as `c-gcc -c -gnatp -emit-header discrec.ads`, CCG produces the following `discrec.h`:

```
#ifndef DISCREC_ADS_H
#define DISCREC_ADS_H

typedef struct discrec__r discrec__r;
struct discrec__r
{
  char dummy_for_null_recordC;
};

typedef struct discrec__TrrS discrec__TrrS;
typedef char ccg_a1[10];
struct discrec__TrrS
{
  int length;
  ccg_a1 s1;
  ccg_a1 s2;
} __attribute__ ((packed));


extern void discrec__rIP (discrec__r *, int);
extern char discrec__first_char (discrec__r *);
extern void discrec___elabs (void);
extern short discrec_E;
extern discrec__TrrS discrec__rr;

#endif /* DISCREC_ADS_H*/
```

When run as `c-gcc -c -gnatp -O2 discrec.ads`, it produces:

```
#include <string.h>
#include <stdlib.h>
```

(continues on next page)

(continued from previous page)

```
typedef struct discrec__r discrec__r;
typedef struct discrec__TrrS discrec__TrrS;
typedef char ccg_a1[10];
struct discrec__TrrS
{
  int length;
  ccg_a1 s1;
  ccg_a1 s2;
} __attribute__ ((packed));


void discrec__rIP (discrec__r *, int);
char discrec__first_char (discrec__r *);
void discrec___elabs (void);
short discrec_E = 0;

inline void discrec__rIP (discrec__r * _init, int length)
{
  *(int *) _init = length;
  return;
}

discrec__TrrS discrec__rr;

inline char discrec__first_char (discrec__r * p)
{
  return ((char *) p)[(((int) ((unsigned int) *(int *) p + 4U)) & 536870911)];
}


void discrec___elabs (void)
{
  discrec__rr.length = 10;
  return;
}
```

To better understand this example, note the following:

- CCG generates a typedef for each array and struct that it encounters. For arrays, it uses names of the form `ccg_a<n>`.
- C has nothing that directly corresponds to a variable-sized discriminated record, so CCG creates:
  - a struct corresponding to the actual type, which is allowed to be incomplete since nothing can ever be of that type
  - a struct corresponding to the fixed initial part of the record, including the discriminant (but only the discriminant in this example).
  - a struct corresponding to each fixed-size instance of the record
- The procedure `discrec__rIP` is an initialization procedure for each instance of the record type (`R`).
- The procedure `discrec___elabs` is an elaboration procedure for the spec of the package, which performs any needed dynamic initialization for the package, which in this case initializes the record `Rr`. An elaboration procedure for the body, if needed, is named `discrec___elabb`. Any such procedures are called from the binder file, which is described in *Calling and Using CCG* (page 7).
- CCG generates significant pointer arithmetic to access the field at a variable position and uses pointer conversions between the structs it created for the record type and the first part of the record.

- The optimized version does less conversions, but has a peculiar logical and operation that's an artifact of the way offset computations are done within a record.

5.6 Type and Object Layout

CCG determines the layout of arrays and records, translates these objects into C arrays and structs, and knows how to compute the offset of each element of an Ada array and each field of an Ada record. It's the responsibility of the C compiler to compute the offset of the corresponding element or field of the C array or struct. These computations should agree.

For the most part, CCG generates C code that mimics the behavior of the Ada code and converts Ada array references into C array references and likewise for field references. For example, consider:

```
package Array_Record is
   type R is record
      F1, F2 : Integer;
   end record;
   type Arr is array (1 .. 10) of R;
   function Get (A : Arr; J : Integer) return Integer is
      (A (J).F2);
end Array_Record;
```

When compiled with `-gnatp`, this produces:

```
#include <string.h>
#include <stdlib.h>

typedef struct array_record__r array_record__r;
struct array_record__r
{
  int f1;
  int f2;
};

typedef array_record__r ccg_a1[10];

int array_record__get (ccg_a1 *, int);
short array_record_E = 0;

inline int array_record__get (ccg_a1 * a, int j)
{
  return (*a)[j - 1].f2;
}
```

Because CCG translated the Ada field and array references into the corresponding C field and array references, that code will operate correctly even if the way CCG lays out arrays and records disagrees with the way your target C compiler does. In simple Ada code, written at an equivalent semantic level to C, this will usually be the case.

But it's not always the case. For example, if you use the `'Position` attribute:

```
package Record_Pos is
   type R is record
      F1, F2 : Integer;
   end record;
   function Get (RR : R; J : Integer) return Integer is
     (RR.F2'Position);
end Record_Pos;
```

This becomes:

```
#include <string.h>
#include <stdlib.h>

typedef struct record_pos__r record_pos__r;
struct record_pos__r
{
  int f1;
  int f2;
};


int record_pos__get (record_pos__r, int);
short record_pos_E = 0;

inline int record_pos__get (record_pos__r rr, int j)
{
  return 4;
}
```

Here, CCG knows that the offset of field `F2` is four bytes from the beginning of record `R`. If you were to then add that value to the value of `'Address` of an object of type `R` and expect that to point to the value of field `F2`, you're assuming your target C compiler lays out `struct record_pos_r` such that field `f2` is four bytes from the beginning of the struct. For this assumption to be valid in general, you need to be sure that CCG knows how your target C compiler lays out structures.

There can be other cases, such as when using other attributes or having more complex layouts, where it's also important that CCG's knowlege of your target compiler is correct. See, for example, the discussions in *Records, Bitfields, and Packing* (page 37).

You can specify the alignments of basic types by providing an LLVM Data Layout String. See *LLVM Data Layout String* (page 22). For composite types, arrays and records, CCG makes the following assumptions about the way your C compiler lays out and aligns data:

- array elements are aligned according to the alignment of their types.

  This means that if the size of an array element isn't a multiple of its alignment (which is unusual), there will be padding bytes between consecutive elements of the array

- the alignment of an array is equal to the alignment of an element of the array
- when a struct that isn't marked `packed` is laid out, the position of each field relative to the start of the struct is a multiple of that field's alignment, adding padding bytes as necessary
- when a packed struct is laid out, fields are laid out in consecutive bytes, with no padding
- the alignment of a struct is the highest alignment of any of its fields if it's not marked `packed` and one byte if it is
- an object is aligned according to the alignment of its type

If one or more of these aren't the case for your target C compiler, please contact AdaCore support.

5.7 Records, Bitfields, and Packing

Ada allows more precise specifications of record layouts than C does. In Ada, we can specify whether a record is packed or not or we can provide a detailed representation of the location of each field. In C, we can specify the width of each field (e.g., a bitfield), but not its position: the only way to force a specific layout is to add explicit padding fields. CCG uses padding to lay out records with record representation clauses.

For example, let's consider the following Ada:

```
package Recrep is
    type R is record
       F1, F2 : Character;
    end record;
    for R use record
       F1 at 0 range 0 .. 7;
       F2 at 2 range 0 .. 7;
    end record;

    RR : R;
 end Recrep;
```

CCG generates the following in `recrep.h`:

```
#ifndef RECREP_ADS_H
#define RECREP_ADS_H

typedef struct recrep__r recrep__r;
struct recrep__r
{
  unsigned char f1;
  char ccg_pad_1;
  unsigned char f2;
};

extern short recrep_E;
extern recrep__r recrep__rr;

#endif /* RECREP_ADS_H*/
```

You can see that CCG generated a padding field, named `ccg_pad_1`, to provide the needed spacing between the fields `F1` and `F2` in the Ada record.

However, CCG doesn't use C bitfields when translating Ada records with small-sized fields. Consider:

```
package Bitrec is
    type R is record
       B1, B2, B3 : Boolean;
       J : Character;
    end record;
    for R use record
       B1 at 1 range 0 .. 0;
       B2 at 1 range 1 .. 1;
       B3 at 1 range 2 .. 2;
       J  at 0 range 0 .. 7;
    end record;

    type Ptr is access all R;
```

(continues on next page)

(continued from previous page)

```
   function Get (P : Ptr) return Character is
      (if P.B2 then P.J else ' ');

end Bitrec;
```

CCG, when invoked as `c-gcc -gnatp -O2 bitrec.ads`, produces:

```
#include <string.h>
#include <stdlib.h>

typedef struct bitrec__r bitrec__r;
struct bitrec__r
{
  unsigned char j;
  char ccg_bits_1;
};

char bitrec__get (bitrec__r *);
short bitrec_E = 0;

inline char bitrec__get (bitrec__r * p)
{
  if ((char) (p->ccg_bits_1 & 2) != 0)
   return p->j;

  return 32;
}
```

In this case, CCG created a field, named `ccg_bits_1`, that contains all the bitfields used in this record. The other field, `j`, corresponds to the field `J` in the original source, which was positioned in front of the bits by the record representation clause.

Most C compilers support both packed and non-packed records, with the default being unpacked. CCG originally lays out all records as packed and adds any needed padding fields. However, it checks whether laying out the record as unpacked will produce the desired field placement (which is the case when you don't specify any representation attributes) and generates an unpacked struct if so. If you've specified that your C compiler doesn't support packing (see `packed-mechanism` in *Target Compiler Parameters* (page 25) for more details), CCG will issue an error message if your program won't execute correctly without that support.

You can see how that works in this case:

```
package Packrec is
   type R1 is record
      F1 : Integer;
      F2, F3, F4, F5 : Character;
   end record with Pack;

   type R2 is record
      F1 : Character;
      F2 : Integer;
      F3, F4, F5 : Character;
   end record with Pack;

   RR1 : R1;
   RR2 : R2;
end Packrec;
```

CCG generates the following in `packrec.h`:

```
#ifndef PACKREC_ADS_H
#define PACKREC_ADS_H

typedef struct packrec__r1 packrec__r1;
struct packrec__r1
{
  int f1;
  unsigned char f2;
  unsigned char f3;
  unsigned char f4;
  unsigned char f5;
};

typedef struct packrec__r2 packrec__r2;
struct packrec__r2
{
  unsigned char f1;
  int f2;
  unsigned char f3;
  unsigned char f4;
  unsigned char f5;
} __attribute__ ((packed));

extern short packrec_E;
extern packrec__r1 packrec__rr1;
extern packrec__r2 packrec__rr2;

#endif /* PACKREC_ADS_H*/
```

Even though both records were specified as packed in the Ada source, the struct corresponding to the first record will have all fields at the same position whether packed or not, so CCG only specifies `packed` for the second record, where the position is different between the packed and unpacked cases.

When CCG produces a packed record, it's possible that your C compiler may produce warnings like the following:

```
pkg1.c:32:20: warning: taking address of packed member of 'struct pkg1__register_PAD_
↪0_' may result in an unaligned pointer value [-Waddress-of-packed-member]
   32 |   ccg_v3 = (int *) &reg_val_.DATA.ccg_bits_0;
```

CCG only generates code like this when it knows that the alignment is such that the pointer will, in fact, be properly aligned, but some C compilers will produce the warnings anyway. In most cases, you can silence the warning with the `-Wno-address-of-packed-member` switch (or the equivalent for your C compiler), as hinted by the warning message.

This page is intentionally left blank.

APPENDIX

A

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

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. Secondarily, 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.