CHAPTER

Using GNU cc

by Kurt Wall

In This Chapter

GNU cc (gcc) is the GNU project's compiler suite. It compiles programs written in C, C++, or Objective C. gcc also compiles Fortran (under the auspices of g77). Front-ends for Pascal, Modula-3, Ada 9X, and other languages are in various stages of development. Because gcc is the cornerstone of almost all Linux development, I will discuss it in some depth. The examples in this chapter (indeed, throughout the book unless noted otherwise), are based on gcc version 2.7.2.3.

Features of GNU cc

gcc gives the programmer extensive control over the compilation process. The compilation process includes up to four stages:

- Preprocessing
- Compilation Proper
- Assembly
- Linking

You can stop the process after any of these stages to examine the compiler's output at that stage. gcc can also handle the various C dialects, such as ANSI C or traditional (Kernighan and Ritchie) C. As noted above, gcc happily compiles C++ and Objective C. You can control the amount and type of debugging information, if any, to embed in the resulting binary and, like most compilers, gcc can also perform code optimization. gcc allows you to mix debugging information and optimization. I strongly discourage doing so, however, because optimized code is hard to debug: Static variables may vanish or loops may be unrolled, so that the optimized program does not correspond line-for-line with the original source code.

gcc includes over 30 individual warnings and three "catch-all" warning levels. gcc is also a cross-compiler, so you can develop code on one processor architecture that will be run on another. Finally, gcc sports a long list of extensions to C and C++. Most of these extensions enhance performance, assist the compiler's efforts at code optimization, or make your job as a programmer easier. The price is portability, however. I will mention some of the most common extensions because you will encounter them in the kernel header files, but I suggest you avoid them in your own code.

A Short Tutorial

Before beginning an in-depth look at gcc, a short example will help you start using gcc productively right away. For the purposes of this example, we will use the program in Listing 3.1.

LISTING 3.1 CANONICAL PROGRAM TO DEMONSTRATE gcc USAGE

```
 1  /*
 2   * Listing 3.1
 3   * hello.c - Canonical "Hello, world!" program 4   4  */
 5  #include <stdio.h>
 6
 7  int main(void)
 8  {
 9      fprintf(stdout, "Hello, Linux programming world!\n");
10      return 0;
11  }
```

To compile and run this program, type

```
$ gcc hello.c -o hello
$ ./hello
Hello, Linux programming world!
```

The first command tells `gcc` to compile and link the source file `hello.c`, creating an executable, specified using the `-o` argument, `hello`. The second command executes the program, resulting in the output on the third line.

A lot took place under the hood that you did not see. `gcc` first ran `hello.c` through the preprocessor, `cpp`, to expand any macros and insert the contents of `#included` files. Next, it compiled the preprocessed source code to object code. Finally, the linker, `ld`, created the `hello` binary.

You can re-create these steps manually, stepping through the compilation process. To tell `gcc` to stop compilation after preprocessing, use `gcc`'s `-E` option:

```
$ gcc -E hello.c -o hello.cpp
```

Examine `hello.cpp` and you can see the contents of `stdio.h` have indeed been inserted into the file, along with other preprocessing tokens. The next step is to compile `hello.cpp` to object code. Use `gcc`'s `-c` option to accomplish this:

```
$ gcc -x cpp-output -c hello.cpp -o hello.o
```

In this case, you do not need to specify the name of the output file because the compiler creates an object filename by replacing `.c` with `.o`. The `-x` option tells `gcc` to begin compilation at the indicated step, in this case, with preprocessed source code.

How does `gcc` know how to deal with a particular kind of file? It relies upon file extensions to determine how to process a file correctly. The most common extensions and their interpretation are listed in Table 3.1.

TABLE 3.1 HOW gcc INTERPRETS FILENAME EXTENSIONS

Extension	*Type*
`.c`	C language source code
`.C`, `.cc`	C++ language source code
`.i`	Preprocessed C source code
`.ii`	Preprocessed C++ source code
`.S`, `.s`	Assembly language source code
`.o`	Compiled object code
`.a`, `.so`	Compiled library code

Linking the object file, finally, creates a binary:

```
$ gcc hello.o -o hello
```

Hopefully, you will see that it is far simpler to use the "abbreviated" syntax we used above, `gcc hello.c -o hello`. I illustrated the step-by-step example to demonstrate that you can stop and start compilation at any step, should the need arise. One situation in which you would want to step through compilation is when you are creating libraries. In this case, you only want to create object files, so the final link step is unnecessary. Another circumstance in which you would want to walk through the compilation process is when an `#included` file introduces conflicts with your own code or perhaps with another `#included` file. Being able to step through the process will make it clearer which file is introducing the conflict.

Most C programs consist of multiple source files, so each source file must be compiled to object code before the final link step. This requirement is easily met. Suppose, for example, you are working on `killerapp.c`, which uses code from `helper.c`. To compile `killerapp.c`, use the following command:

```
$ gcc killerapp.c helper.c -o killerapp
```

`gcc` goes through the same preprocess-compile-link steps as before, this time creating object files for each source file before creating the binary, `killerapp`. Typing long commands like this does become tedious. In Chapter 4, "Project Management Using GNU `make`," we will see how to solve this problem. The next section will begin introducing you to the multitude of `gcc`'s command-line options.

Common Command-line Options

The list of command-line options gcc accepts runs to several pages, so we will only look at the most common ones in Table 3.2.

TABLE 3.2 gcc COMMAND-LINE OPTIONS

Option	*Description*
`-o FILE`	Specify the output filename; not necessary when compiling to object code. If `FILE` is not specified, the default name is `a.out`.
`-c`	Compile without linking.
`-DFOO=BAR`	Define a preprocessor macro named `FOO` with a value of `BAR` on the command-line.
`-IDIRNAME`	Prepend `DIRNAME` to the list of directories searched for include files.
`-LDIRNAME`	Prepend `DIRNAME` to the list of directories searched for library files. By default, `gcc` links against shared libraries.
`-static`	Link against static libraries.
`-lFOO`	Link against `libFOO`.
`-g`	Include standard debugging information in the binary.
`-ggdb`	Include lots of debugging information in the binary that only the GNU debugger, gdb, can understand.
`-O`	Optimize the compiled code.
`-ON`	Specify an optimization level N, 0<=N<= 3.
`-ansi`	Support the ANSI/ISO C standard, turning off GNU extensions that conflict with the standard (this option does not guarantee ANSI-compliant code).
`-pedantic`	Emit all warnings required by the ANSI/ISO C standard.
`-pedantic-errors`	Emit all errors required by the ANSI/ISO C standard.
-traditional	Support the Kernighan and Ritchie C language syntax (such as the old-style function definition syntax). If you don't understand what this means, don't worry about it.
`-w`	Suppress all warning messages. In my opinion, using this switch is a very bad idea!

continues

TABLE 3.2 CONTINUED

Option	*Description*
`-Wall`	Emit all generally useful warnings that `gcc` can provide. Specific warnings can also be flagged using `-W{warning}`.
`-werror`	Convert all warnings into errors, which will stop the compilation.
`-MM`	Output a `make`-compatible dependency list.
`-v`	Show the commands used in each step of compilation.

We have already seen how `-c` works, but `-o` needs a bit more discussion. `-o FILE` tells `gcc` to place output in the file `FILE` regardless of the output being produced. If you do not specify `-o`, the defaults for an input file named `FILE.SUFFIX` are to put an executable in `a.out`, object code in `FILE.o`, and assembler code in `FILE.s`. Preprocessor output goes to standard output.

Library and Include Files

If you have library or include files in non-standard locations, the `-L{DIRNAME}` and `-I{DIRNAME}` options allow you to specify these locations and to insure that they are searched before the standard locations. For example, if you store custom include files in `/usr/local/include/killerapp`, then in order for `gcc` to find them, your `gcc` invocation would be something like

```
$ gcc someapp.c -I/usr/local/include/killerapp
```

Similarly, suppose you are testing a new programming library, `libnew.so` (`.so` is the normal extension for shared libraries— more on this subject in Chapter 24, "Using Libraries") currently stored in `/home/fred/lib`, before installing it as a standard system library. Suppose also that the header files are stored in `/home/fred/include`. Accordingly, to link against `libnew.so` and to help `gcc` find the header files, your `gcc` command line should resemble the following:

```
$gcc myapp.c -L/home/fred/lib -I/home/fred/include -lnew
```

The `-l` option tells the linker to pull in object code from the specified library. In this example, I wanted to link against `libnew.so`. A long-standing UNIX convention is that libraries are named lib{something}, and `gcc`, like most compilers, relies on this convention. If you fail to use the `-l` option when linking against libraries, the link step will fail and `gcc` will complain about undefined references to "function_name."

By default, gcc uses shared libraries, so if you must link against static libraries, you have to use the -static option. This means that only static libraries will be used. The following example creates an executable linked against the static ncurses. Chapter 27, "Screen Manipulation with ncurses," discusses user interface programming with ncurses:

```
$ gcc cursesapp.c -lncurses -static
```

When you link against static libraries, the resulting binary is much larger than using shared libraries. Why use a static library, then? One common reason is to guarantee that users can run your program—in the case of shared libraries, the code your program needs to run is linked dynamically at runtime, rather than statically at compile time. If the shared library your program requires is not installed on the user's system, she will get errors and not be able to run your program.

The Netscape browser is a perfect example of this. Netscape relies heavily on Motif, an expensive X programming toolkit. Most Linux users cannot afford to install Motif on their system, so Netscape actually installs two versions of their browser on your system; one that is linked against shared libraries, netscape-dynMotif, and one that is statically linked, netscape-statMotif. The netscape "executable" itself is actually a shell script that checks to see if you have the Motif shared library installed and launches one or the other of the binaries as necessary.

Error Checking and Warnings

gcc boasts a whole class of error-checking, warning-generating, command-line options. These include -ansi, -pedantic, -pedantic- errors, and -Wall. To begin with, -pedantic tells gcc to issue all warnings demanded by strict ANSI/ISO standard C. Any program using forbidden extensions, such as those supported by gcc, will be rejected. -pedantic-errors behaves similarly, except that it emits errors rather than warnings. -ansi, finally, turns off GNU extensions that do not comply with the standard. None of these options, however, guarantee that your code, when compiled without error using any or all of these options, is 100 percent ANSI/ISO-compliant.

Consider Listing 3.2, an example of very bad programming form. It declares main() as returning void, when in fact main() returns int, and it uses the GNU extension long long to declare a 64-bit integer.

LISTING 3.2 NON-ANSI/ISO SOURCE CODE

```
1  /*
2   * Listing 3.2
3   * pedant.c - use -ansi, -pedantic or -pedantic-errors
```

continues

LISTING 3.2 CONTINUED

```
 4  */
 5  #include <stdio.h>
 6
 7  void main(void)
 8  {
 9      long long int i =  0l;
10      fprintf(stdout, "This is a non-conforming C program\n");
11  }
```

Using `gcc pedant.c -o pedant`, this code compiles without complaint. First, try to compile it using `-ansi`:

```
$ gcc -ansi pedant.c -o pedant
```

Again, no complaint. The lesson here is that `-ansi` forces `gcc` to emit the diagnostic messages required by the standard. It does not insure that your code is ANSI C[nd]compliant. The program compiled despite the deliberately incorrect declaration of `main()`. Now, `-pedantic`:

```
$ gcc -pedantic pedant.c -o pedant
pedant.c: In function `main':
pedant.c:9: warning: ANSI C does not support `long long'
```

The code compiles, despite the emitted warning. With `-pedantic- errors`, however, it does not compile. `gcc` stops after emitting the error diagnostic:

```
$ gcc -pedantic-errors pedant.c -o pedant
pedant.c: In function `main':
pedant.c:9: ANSI C does not support `long long'
$ ls
a.out*       hello.c      helper.h     killerapp.c
hello*       helper.c     killerapp*   pedant.c
```

To reiterate, the `-ansi`, `-pedantic`, and `-pedantic-errors` compiler options do not insure ANSI/ISO-compliant code. They merely help you along the road. It is instructive to point out the remark in the `info` file for `gcc` on the use of `-pedantic`:

"This option is not intended to be useful; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard. Some users try to use `-pedantic' to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all—only those for which ANSI C *requires* a diagnostic."

Optimization Options

Code optimization is an attempt to improve performance. The trade-off is lengthened compile times and increased memory usage during compilation.

The bare `-O` option tells `gcc` to reduce both code size and execution time. It is equivalent to `-O1`. The types of optimization performed at this level depend on the target processor, but always include at least thread jumps and deferred stack pops. Thread jump optimizations attempt to reduce the number of jump operations; deferred stack pops occur when the compiler lets arguments accumulate on the stack as functions return and then pops them simultaneously, rather than popping the arguments piecemeal as each called function returns.

`O2` level optimizations include all first-level optimization plus additional tweaks that involve processor instruction scheduling. At this level, the compiler takes care to make sure the processor has instructions to execute while waiting for the results of other instructions or data latency from cache or main memory. The implementation is highly processor-specific. `-O3` options include all `O2` optimizations, loop unrolling, and other processor-specific features.

Depending on the amount of low-level knowledge you have about a given CPU family, you can use the `-f{flag}` option to request specific optimizations you want performed. Three of these flags bear consideration: `-ffastmath`, `-finline-functions`, and `-funroll-loops`. `-ffastmath` generates floating-point math optimizations that increase speed, but violate IEEE and/or ANSI standards. `-finline-functions` expands all "simple" functions in place, much like preprocessor macro replacements. Of course, the compiler decides what constitutes a simple function. `-funroll-loops` instructs `gcc` to unroll all loops that have a fixed number of iterations that can be determined at compile time. Inlining and loop unrolling can greatly improve a program's execution speed because they avoid the overhead of function calls and variable lookups, but the cost is usually a large increase in the size of the binary or object files. You will have to experiment to see if the increased speed is worth the increased file size. See the `gcc` info pages for more details on processor flags.

NOTE

For general usage, using `-O2` optimization is sufficient. Even on small programs, like the `hello.c` program introduced at the beginning of this chapter, you will see small reductions in code size and small increases in performance time.

Debugging Options

Bugs are as inevitable as death and taxes. To accommodate this sad reality, use `gcc`'s `-g` and `-ggdb` options to insert debugging information into your compiled programs to facilitate debugging sessions.

The `-g` option can be qualified with a 1, 2, or 3 to specify how much debugging information to include. The default level is 2 (`-g2`), which includes extensive symbol tables, line numbers, and information about local and external variables. Level 3 debugging information includes all of the level 2 information and all of the macro definitions present. Level 1 generates just enough information to create backtracks and stack dumps. It does not generate debugging information for local variables or line numbers.

If you intend to use the GNU Debugger, `gdb` (covered in Chapter 36, "Debugging: GNU `gdb`"), using the `-ggdb` option creates extra information that eases the debugging chore under `gdb`. However, this will also likely make the program impossible to debug using other debuggers, such as the `DBX` debugger common on the Solaris operating system. `-ggdb` accepts the same level specifications as `-g`, and they have the same effects on the debugging output. Using either of the two debug-enabling options will, however, dramatically increase the size of your binary. Simply compiling and linking the simple `hello.c` program I used earlier in this chapter resulted in a binary of 4089 bytes on my system. The resulting sizes when I compiled it with the `-g` and `-ggdb` options may surprise you:

```
$ gcc -g hello.c -o hello_g
$ ls -l hello_g
-rwxr-xr-x   1 kwall    users        6809 Jan 12 15:09 hello_g*

$ gcc -ggdb hello.c -o hello_ggdb
$ ls -l hello_ggdb
-rwxr-xr-x   1 kwall    users      354867 Jan 12 15:09
hello_ggdb*
```

As you can see, the `-g` option increased the binary's size by half, while the `-ggdb` option bloated the binary nearly 900 percent! Despite the size increase, I recommend shipping binaries with standard debugging symbols (created using `-g`) in them in case someone encounters a problem and wants to try to debug your code for you.

Additional debugging options include the `-p` and `-pg` options, which embed profiling information into the binary. This information is useful for tracking down performance bottlenecks in your code. `-p` adds profiling symbols that the `prof` program can read, and `-pg` adds symbols that the GNU project's `prof` incarnation, `gprof`, can interpret. The `-a` option generates counts of how many times blocks of code (such as functions) are entered. `-save-temps` saves the intermediate files, such as the object and assembler files, generated during compilation.

Finally, as I mentioned at the beginning of this chapter, `gcc` allows you simultaneously to optimize your code and insert debugging information. Optimized code presents a debugging challenge, however, because variables you declare and use may not be used in the optimized program, flow control may branch to unexpected places, statements that compute constant values may not execute, and statements inside loops will execute elsewhere because the loop was unrolled. My personal preference, though, is to debug a program thoroughly before worrying about optimization. Your mileage may vary.

NOTE

Do not, however, take "optimize later" to mean "ignore efficiency during the design process." Optimization, in the context of this chapter, refers to the compiler magic I have discussed in this section. Good design and efficient algorithms have a far greater impact on overall performance than any compiler optimization ever will. Indeed, if you take the time up front to create a clean design and use fast algorithms, you may not need to optimize, although it never hurts to try.

GNU C Extensions

GNU C extends the ANSI standard in a variety of ways. If you don't mind writing blatantly non-standard code, some of these extensions can be very useful. For all of the gory details, I will direct the curious reader to `gcc`'s info pages. The extensions covered in this section are the ones frequently seen in Linux's system headers and source code.

To provide 64-bit storage units, for example, `gcc` offers the "long long" type:

```
long long long_int_var;
```

NOTE

The "long long" type exists in the new draft ISO C standard.

On the x86 platform, this definition results in a 64-bit memory location named `long_int_var`. Another `gcc`-ism you will encounter in Linux header files is the use of inline functions. Provided it is short enough, an inline function expands in your code much as a macro does, thus eliminating the cost of a function call. Inline functions are better than macros, however, because the compiler type-checks them at compile time. To use the inline functions, you have to compile with at least `-O` optimization.

The `attribute` keyword tells `gcc` more about your code and aids the code optimizer. Standard library functions, such as `exit()` and `abort()`, never return so the compiler can generate slightly more efficient code if it knows that the function does not return. Of course, userland programs may also define functions that do not return. `gcc` allows you to specify the `noreturn` attribute for such functions, which acts as a hint to the compiler to optimize the function.

Suppose, for example, you have a function named `die_on_error()` that never returns. To use a function attribute, append `__attribute__ ((attribute_name))` after the closing parenthesis of the function declaration. Thus, the declaration of `die_on_error()` would look like:

```
void die_on_error(void) __attribute__ ((noreturn));
```

The function would be defined normally:

```
void die_on_error(void)
{
     /* your code here */
     exit(1);
}
```

You can also apply attributes to variables. The `aligned` attribute instructs the compiler to align the variable's memory location on a specified byte boundary.

```
int int_var __attribute__ ((aligned 16)) = 0;
```

will cause `gcc` to align `int_var` on a 16-byte boundary. The `packed` attribute tells `gcc` to use the minimum amount of space required for variables or structs. Used with `structs`, `packed` will remove any padding that `gcc` would ordinarily insert for alignment purposes.

A terrifically useful extension is case ranges. The syntax looks like:

```
case LOWVAL ... HIVAL:
```

Note that the spaces preceding and following the ellipsis are required. Case ranges are used in `switch()` statements to specify values that fall between LOWVAL and HIVAL:

```
switch(int_var) {
     case 0 ... 2:
        /* your code here */
        break;
     case 3 ... 5:
        /* more code here */
        break;
     default:
        /* default code here */
}
```

The preceding fragment is equivalent to:

```
switch(int_var) {
     case 1:
     case 2:
        /* your code here */
        break;
     case 3:
     case 4:
     case 5:
        /* more code here */
        break;
     default:
        /* default code here */
}
```

Case ranges are just a shorthand notation for the traditional `switch()` statement syntax.

Summary

In this chapter, I have introduced you to `gcc`, the GNU compiler suite. In reality, I have only scratched the surface, though; `gcc`'s own documentation runs to several hundred pages. What I have done is show you enough of its features and capabilities to enable you to start using it in your own development projects.