THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to GCC

Everything you need to know about using the GNU Compiler Collection and related tools

SECOND EDITION

William von Hagen

Apress®

The Definitive Guide to GCC

Second Edition

William von Hagen

Apress®

The Definitive Guide to GCC, Second Edition

ISBN-13 (pbk): 978-1-59059-585-5

ISBN-10 (pbk): 1-59059-585-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Jason Gilmore, Keir Thomas
Technical Reviewer: Gene Sally
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Elizabeth Berry
Indexer: Toma Mulligan
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

To Dorothy Fisher, for all your love, support, and encouragement.
And for Becky Gable—what would we do without the schematics?
—Bill von Hagen

Contents at a Glance

Contents

About the Author

BILL VON HAGEN holds degrees in computer science, English writing, and art history. Bill has worked with Unix systems since 1982, during which time he has been a system administrator, writer, systems programmer, development manager, drummer, operations manager, content manager, and product manager. Bill has written a number of books including *The Ubuntu Bible*, *Hacking the TiVo*, *Linux Filesystems*, *Installing Red Hat Linux*, and *SGML for Dummies*; coauthored *Linux Server Hacks, Volume 2* and *Mac OS X Power User's Guide*; and contributed to several other books. Bill has written articles and software reviews for publications including *Linux Journal*, *Linux Magazine*, *Mac Tech*, *Linux Format* (UK), *Mac Format* (UK), and *Mac Directory*. He has also written extensive online content for CMP Media, Linux Planet, and Linux Today. An avid computer collector specializing in workstations, he owns more than 200 computer systems. You can contact Bill at wvh@vonhagen.org.

About the Technical Reviewer

GENE SALLY has been a Linux enthusiast for the past ten years, and for the past six he has channeled his enthusiasm through his employer, TimeSys, creating tools for embedded Linux engineers and helping them become more productive. Embedded development pushes the envelope of most technologies, Linux and GCC included, so Gene has had the opportunity to push these tools to their limits as he creates development tools and technologies for TimeSys' customers.

Acknowledgments

I'd like to thank Kurt Wall for his friendship and the opportunity to work with him on the first edition of this book, and Marta Justak, of Justak Literary Services, for her support and help with this book. I'd also like to thank Gene Sally for making this book far better than it could have been without him, and Richard Dal Porto, Keir Thomas, Jason Gilmore, Jennifer Whipple, Katie Stence, and others at Apress for their patience (!) and support for this second edition. In general, I'd like to thank GCC, emacs (the one true editor), Richard Stallman and the FSF, 50 million BSD fans (who can't be wrong), and Linux Torvalds and a cast of thousands for their contributions to computing as we know it today.

Without their foresight, philosophy, and hard work, this book wouldn't even exist. I'd especially like to thank rms for some way cool LMI hacks long ago.

Introduction

This book, *The Definitive Guide to GCC,* is about how to build, install, customize, use, and troubleshoot GCC version 4.x. GCC has long been available for most major hardware and operating system platforms and is often the preferred family of compilers.

As a general-purpose set of compilers, GCC produces high-quality, fast code. Due to its design, GCC is easy to port to different architectures, which contributes to its popularity. GCC, along with GNU Emacs, the Linux operating system, the Apache Web server, the Sendmail mail server, and the BIND DNS server, are showpieces of the free software world and proof that sometimes you **can** get a free lunch.

Why a Book About GCC?

I wrote this book, and you should read it, for a variety of reasons: it covers version 4.x; it is the only book that covers general GCC usage; and I would argue that it is better than GCC's own documentation. You will not find more complete coverage of GCC's features, quirks, and usage anywhere else in a single volume. There are no other up-to-date sources of information on GCC, excluding GCC's own documentation. GCC usually gets one or two chapters in programming books and only a few paragraphs in other more general titles.

GCC's existing documentation, although thorough and comprehensive, targets a programming-savvy reader. There's certainly nothing wrong with this approach, which is certainly the proper approach for advanced users, but GCC's own documentation leaves the great majority of its users out in the cold. Much of *The Definitive Guide to GCC* is tutorial and practical in nature, explaining why you use one option or why you should not use another one. In addition, explaining auxiliary tools and techniques that are relevant to GCC but not explicitly part of the package helps make this book a complete and usable guide and reference. Showing you how to use the compilers in the GCC family and related tools, and helping you get your work done are this book's primary goals.

Most people, including many long-time programmers, use GCC the way they learned or were taught to use it. That is, many GCC users treat the compiler as a black box, which means that they invoke it by using a small and familiar set of options and arguments they have memorized, shoving source files in one end, and then receiving a compiled, functioning program from the other end. With a powerful set of compilers such as GCC, there are indeed stranger (and more useful) things than were dreamed of in Computer Science 101. Therefore, another goal when writing *The Definitive Guide to GCC* was to reveal cool but potentially obscure options and techniques that you may find useful when building or using GCC and related tools and libraries.

Inveterate tweakers, incorrigible tinkerers, and the just plain adventurous among you will also enjoy the chance to play with the latest and greatest version of GCC and the challenge of bending a complex piece of software to your will, especially if you have instructions that show you how to do so with no negative impact on your existing system.

Why the New Edition?

I've written a new edition of this book for two main reasons: much has changed in GCC since the first edition of this book came out, and I wanted to talk about the other GCC compilers and related technologies such as cross-compilers and alternate C libraries. The GCC 4.x family of compilers is now available, providing a new optimization framework, many associated improvements to optimization in general, a new Fortran compiler, significant performance improvements for the C++ compiler, huge updates to the Java compiler, just-in-time compilation for Java, support for many new platforms, and enough new options in general to keep you updating Makefiles for quite a while. The first edition of this book focused on the C and C++ compilers in GCC, but enquiring minds want to know much more. This edition substantially expands the C++ coverage and adds information about using the Fortran, Java, and Objective-C compilers. No one has ever asked me about the Ada compiler, so I've still skipped that one. In addition, I've added information on using alternate C libraries and building cross-compilers that should make this book more valuable to its existing audience and (hopefully) attractive to an even larger one.

What You Will Learn

The Definitive Guide to GCC now provides a chapter dedicated to explaining how to use each of the C, C++, Fortran, and Java compilers. Information that is common to all of the compilers has been moved to Appendix A, so as not to repeat it everywhere and keep you from getting started with your favorite compiler. Similarly, information about building GCC has been moved to much later in the book, since most readers simply want to use the compilers that they find on their Linux and *BSD systems, not necessarily build them from scratch. However, if you want the latest and greatest version of GCC, you will learn how to download, compile, and install GCC from scratch, a poorly understood procedure that, until now, only the most capable and confident users have been willing to undertake.

The chapter on troubleshooting compilation problems has been expanded to make it easier than ever to discover problems in your code or the configuration or installation of your GCC compilers. If you're a traditional Makefile fan, the chapters on Libtool, Autoconf, and Automake will help you produce your Makefiles automatically, making it easier to package, archive, and distribute the source code for your projects. The chapters on code optimization, test coverage, and profiling have been expanded and updated to discuss the latest techniques and tools, helping you debug, improve, and test your code more extensively than ever. Finally, the book veers back to its focus for a more general audience by providing a complete summary of the GCC's command-line interface, a chapter on troubleshooting GCC usage and installation, and another chapter explaining how to use GCC's online documentation.

What You Need to Know

This is an end user's book intended for anyone using almost all of the GCC compilers (sorry, Ada fans). Whether you are a casual end user who only occasionally compiles programs, an intermediate user using GCC frequently but lacking much understanding of how it works, or a programmer seeking to exercise GCC to the full extent of its capabilities, you will find information in this book that you can use immediately. Because Linux and Intel x86 CPUs are so popular, I've assumed that most of you are using one version or another of the Linux operating system running on Intel x86 or compatible systems. This isn't critical—most of the material is GCC-specific, rather than being Linux- or Intel-specific, because GCC is largely independent of operating systems and CPU features in terms of its usage.

What do you need to know to benefit from this book? Well, knowing how to type is a good start because the GCC compilers are command-line compilers. (Though GCC compilers are integrated

into many graphical integrated development environments, that is somewhat outside the scope of this book.) You should therefore be comfortable with working in a command-line environment, such as a terminal window or a Unix or Linux console. You need to be computer literate, and the more experience you have with Unix or Unix-like systems, such as Linux, the better. If you have downloaded and compiled programs from source code before, you will be familiar with the terminology and processes discussed in the text. If, on the other hand, this is your first foray into working with source code, the chapters on building GCC and C libraries will get you up and running quickly. You do not need to be a programming wizard or know how to do your taxes in hexadecimal. Any experience that you have using a compiled programming language is gravy.

You should also know how to use a text editor, such as vi, pico, or Emacs, if you intend to type the listings and examples yourself in order to experiment with them. Because the source and binary versions of the GCC are usually available in some sort of compressed format, you will also need to know how to work with compressed file formats, usually gzipped tarballs, although the text will explain how to do so.

What *The Definitive Guide to GCC* Does Not Cover

As an end user's book on GCC, a number of topics are outside this book's scope. In particular, it is not a primer on C, C++, Fortran, or Java, although each chapter provides a consistent set of programming examples that I've used throughout the book. As discussed throughout this book, GCC is a collection of front-end, language-specific interfaces to a common back-end compilation engine. The list of compilers includes C, C++, Objective C, Fortran, Ada, and Java, among others. Compiler theory gets short shrift in this book, because I believe that most people are primarily interested in getting work done with GCC, not writing it. The Free Software Foundation has some excellent documents on GCC internals on its Web site, and it doesn't get much more definitive than that. That said, it is difficult to talk about using a compiler without skimming the surface of compiler theory and operation, so this book defines key terms and concepts as necessary while describing GCC's architecture and overall compilation workflow.

History and Overview of GCC

This section takes a more thorough look at what GCC is and does and includes the obligatory history of GCC. Because GCC is one of the GNU Project's premier projects, GCC's development model bears a closer look, so I will also show you GCC's development model, which should help you understand why GCC has some features and lacks other features, and how you can participate in its development.

What exactly is GCC? The tautological answer is that GCC is an acronym for the GNU Compiler Collection, formerly known as the GNU Compiler Suite, and also known as GNU CC and the GNU C Compiler. As remarked earlier, GCC is a collection of compiler front ends to a common back-end compilation engine. The list of compilers includes C, C++, Objective C, Fortran (now 95, formerly 77), and Java. GCC also has front ends for Pascal, Modula-3, and Ada 9X. The C compiler itself speaks several different dialects of C, including traditional and ANSI C. The C++ compiler is a true native C++ compiler. That is, it does not first convert C++ code into an intermediate C representation before compiling it, as did the early C++ compilers such as the Cfront "compiler" Bjarne Stroustrup first used to create C++. Rather, GCC's C++ compiler, g++, creates native executable code directly from the C++ source code.

GCC is an optimizing and cross-platform compiler. It supports general optimizations that can be applied regardless of the language in use or the target CPU and options specific to particular CPU families and even specific to a particular CPU model within a family of related processors. Moreover, the range of hardware platforms to which GCC has been ported is remarkably long. GCC supports platform and target submodels, so that it can generate executable code that will run on all members

of a particular CPU family or only on a specific model of that family. Table 1 provides a partial list of GCC's supported architectures, many of which you might never have heard of, much less used. Frankly, I haven't used (or even seen) all of them. For a more definitive list, see Appendix B, which summarizes architectures and processor-specific options for your convenience.

Considering the variety of CPUs and architectures to which GCC has been ported, it should be no surprise that you can configure it as a cross-compiler and use GCC to compile code on one platform that is intended to run on an entirely different platform. In fact, you can have multiple GCC configurations for various platforms installed on the same system and, moreover, run multiple GCC versions (older and newer) for the same CPU family on the same system.

Table 1. *Some of the Most Popular Processor Architectures Supported by GCC*

Architecture	Description
AMD29K	AMD Am29000 architectures
AMD64	64-bit AMD processors that are compatible with the Intel-32 architecture
ARM	Advanced RISC Machines architectures
ARC	Argonaut ARC processors
AVR	Atmel AVR microcontrollers
ColdFire	Motorola's latest generation of 68000 descendents
DEC Alpha	Compaq (neé Digital Equipment Corporation) Alpha processors
H8/300	Hitachi H8/300 CPUs
HP/PA	Hewlett-Packard PA-RISC architectures
Intel i386	Intel i386 (x86) family of CPUs
Intel i960	Intel i960 family of CPUs
M32R/D	Mitsubishi M32R/D architectures
M68K	The Motorola 68000 series of CPUs
M88K	Motorola 88K architectures
MCore	Motorola M*Core processors
MIPS	MIPS architectures
MN10200	Matsushita MN10200 architectures
MN10300	Matsushita MN10300 architectures
NS32K	National Semiconductor NS3200 CPUs
RS/6000 and PowerPC	IBM RS/6000 and PowerPC architectures
S390	IBM processors used in zSeries and System z mainframe
SPARC	Sun Microsystems family of SPARC CPUs
SH3/4/5	Super Hitachi 3, 4, and 5 family of processors
TMS320C3x/C4x	Texas Instruments TMS320C3x and TMS320C4x DSPs

GCC's History

GCC, or rather, the idea for it, actually predates the GNU Project. In late 1983, just before he started the GNU Project, Richard M. Stallman, president of the Free Software Foundation and originator of the GNU Project, heard about a compiler named the Free University Compiler Kit (known as VUCK) that was designed to compile multiple languages, including C, and to support multiple target CPUs. Stallman realized that he needed to be able to bootstrap the GNU system and that a compiler was the first strap he needed to boot. So he wrote to VUCK's author asking if GNU could use it. Evidently, VUCK's developer was uncooperative, responding that the university was free but that the compiler was not. As a result, Stallman concluded that his first program for the GNU Project would be a multilanguage, cross-platform compiler. Undeterred and in true hacker fashion, desiring to avoid writing the entire compiler himself, Stallman eventually obtained the source code for Pastel, a multiplatform compiler developed at Lawrence Livermore National Laboratory. He added a C front end to Pastel and began porting it to the Motorola 68000 platform, only to encounter a significant technical obstacle: the compiler's design required many more megabytes of stack space than the 68000-based Unix system supported. This situation forced him to conclude that he would have to write a new compiler, starting from ground zero. That new compiler eventually became GCC.

Although it contains none of the Pastel source code that originally inspired it, Stallman did adapt and use the C front end he wrote for Pastel. As a starting point for GCC's optimizer, Stallman also used PO, a portable peephole optimizer that performed optimizations generally done by high-level optimizers, in addition to low-level peephole optimizers. GCC (and PO's successor, vpo) still uses RTL (register transfer language) as an intermediate format for the optimizer. Development of this primordial GCC proceeded slowly through the 1980s, because, as Stallman writes in his description of the GNU Project (`http://www.gnu.org/gnu/the-gnu-project.html`), "first, [he] worked on GNU Emacs."

During the 1990s, GCC development split into two, perhaps three, branches. While the primary GCC branch continued to be maintained by the GNU Project, a number of other developers, primarily associated with Cygnus Solutions, began releasing a version of GCC known as EGCS (Experimental [or Enhanced] GNU Compiler Suite). EGCS was intended to be a more actively developed and more efficient compiler than GCC, but was otherwise effectively the same compiler because it closely tracked the GCC code base and EGCS enhancements were fed back into the GCC code base maintained by the GNU Project. Nonetheless, the two code bases were separately maintained. In April 1999, GCC's maintainers, the GNU Project, and the EGCS steering committee formally merged. At the same time, GCC's name was changed to the GNU Compiler Collection and the separately maintained (but, as noted, closely synchronized) code trees were formally combined, ending a long fork and incorporating the many bug fixes and enhancements made in EGCS into GCC. This is why EGCS is often mentioned, though it is officially defunct.

Other historical variants of GCC include the Pentium Compiler Group (PCG) project's own version of GCC, PGCC. PGCC was a Pentium-specific version that was intended to provide the best possible support for features found in Intel's Pentium-class CPUs. During the period of time that EGCS was separately maintained, PGCC closely tracked the EGCS releases. The reunification of EGCS and GCC seems to have halted PGCC development because, at the time of this writing, the PCG project's last release was 2.95.2.1, dated December 27, 2000. For additional information, visit the PGCC project's Web site at `http://www.goof.com/pcg/`.

At the time that this book was written, GCC 4.2 was about to become available. The latest officially released version of the GCC 3.x line of compilers is 3.4.5. Other significant milestone compilers are the 2.95.x compilers, which were widely hacked to produce code for a variety of embedded systems and which are still widely available.

Who Maintains GCC?

Formally, GCC is a GNU Project, which is directed by the FSF. The FSF holds the copyright on the compilers, and licenses the compilers under the terms of the GPL. Either individuals or the FSF hold

the copyrights on other components, such as the runtime libraries and test suites, and these other components are licensed under a variety of licenses for free software. For information on the licensing of any FSF package see the file LICENSE that is provided with its source code distribution. The FSF also handles the legal concerns of the GCC project. So much for the administrivia.

On the practical side, a cast of dozens maintains GCC. GCC's maintainers consist of a formally organized steering committee and a larger, more loosely organized group of hackers scattered all over the Internet. The GCC steering committee, as of August 2001, is made up of 14 people representing various communities in GCC's user base who have a significant stake in GCC's continuing and long-term survival, including kernel hackers, Fortran users, and embedded systems developers. The steering committee's purpose is, to quote its mission statement, "to make major decisions in the best interests of the GCC project and to ensure that the project adheres to its fundamental principles found in the project's mission statement." These "fundamental principles" include the following:

- Supporting the goals of the GNU Project
- Adding new languages, optimizations, and targets to GCC
- More frequent releases
- Greater responsiveness to consumers, the large user base that relies on the GCC compiler
- An open development model that accepts input and contributions based on technical merit

The group of developers that work on GCC includes members of the steering committee and, according to the contributors list on the GCC project home page, more than 100 other individuals across the world. Still, others not specifically identified as contributors have contributed to GCC development by sending in patches, answering questions on the various GCC mailing lists, submitting bug reports, writing documentation, and testing new releases.

Who Uses GCC?

GCC's user base is large and varied. Given the nature of GCC and the loosely knit structure of the free software community, though, no direct estimate of the total number of GCC users is possible. A direct estimate, based on standard metrics, such as sales figures, unit shipments, or license purchases, is virtually impossible to derive because such numbers simply do not exist. Even indirect estimates, based, for example, on the number of downloads from the GNU Web and FTP sites, would be questionable because the GNU software repository is mirrored all over the world.

More to the point, I submit that quantifying the number of GCC users is considerably less important and says less about GCC users than examining the scope of GCC's usage and the number of processor architectures to which it has been ported. For example, GCC is the standard compiler shipped in every major and most minor Linux distributions. GCC is also the compiler of choice for the various BSD operating systems (FreeBSD, NetBSD, OpenBSD, and so on). Thanks initially to the work of DJ Delorie, GCC works on most modern DOS versions, including MS-DOS from Microsoft, PC-DOS from IBM, and DR-DOS. Indeed, Delorie's work resulted in ports of most of the GNU tools for DOS-like environments. Cygnus Solutions, now owned by Red Hat, Inc., created a GCC port for Microsoft Windows users. Both the DOS and Windows ports offer complete and free development environments for DOS and Windows users.

The academic computing community represents another large part of GCC's user base. Vendors of hardware and proprietary operating systems typically provide compiler suites for their products as a so-called value-added service, that is, for an additional, often substantial, charge. As free software, GCC represents a compelling, attractive alternative to computer science departments faced with tight budgets. GCC also appeals to the academic world because it is available in source code form, giving students a chance to study compiler theory, design, and implementation. GCC is also widely used by nonacademic customers of hardware and operating system vendors who want to

reduce support costs by using a free, high-quality compiler. Indeed, if you consider the broad range of hardware to which GCC has been ported, it becomes quite clear that GCC's user base is composed of the broadest imaginable range of computer users.

In general, my favorite response from any reader of this book to the question of who uses GCC is "I do."

Are There Alternatives?

What alternatives to GCC exist? As framed, this question is somewhat difficult to answer. Remember that GCC is the GNU Compiler Collection, a group of language-specific compiler front ends using a common back-end compilation engine, and that GCC is free software. So if you rephrase the question to "what free compiler suites exist as alternatives to GCC?" the answer is "very few."

As mentioned earlier, the Pentium Compiler Group created PGCC, a version of GCC, that was intended to extend GCC's ability to optimize code for Intel's Pentium-class CPUs. Although PGCC development seems to have stalled since the EGCS/GCC schism ended, the PGCC Web site still exists (although it, too, has not been modified recently).

If you remove the requirement that the alternative be free, you have many more options. Many hardware vendors and most operating system vendors will be happy to sell you compiler suites for their respective hardware platforms or operating systems, but the cost can be prohibitive. Some third-party vendors exist that provide stand-alone compiler suites. One such vendor is The Portland Group (`http://www.pgroup.com/`), which markets a set of high-performance, parallelizing compiler suites supporting Fortran, C, and C++. Absoft Corporation also offers a well-regarded compiler suite supporting Fortran 77, Fortran 95, C, and C++. Visit its Web site at `http://www.absoft.com/` for additional information. Similarly, Borland has a free C/C++ compiler available. Information on Borland's tools can be found on its Web site at `http://www.borland.com/`. Intel and Microsoft also sell very good compilers. And they are not that expensive.

Conversely, if you dispense with the requirement that alternatives be collections or suites, you can select from a rich array of options. A simple search for the word *compilers* at Yahoo! generates more than 120 Web sites showcasing a variety of single-language compilers, including Ada, Basic, C and C++, COBOL, Forth, Java, Logo, Modula-2 and Modula-3, Pascal, Prolog, and Smalltalk. If you are looking for alternatives to GCC, a good place to start your search is the compilers.net Web page at `http://www.compilers.net/`.

So much for a look at alternatives to GCC. This is a book about GCC, after all, so I hope that you'll forgive me for largely leaving you on your own when it comes to finding information about other compilers. Some chapters of this book, such as the chapter on the new GCC Fortran compiler, gfortran, discuss alternatives because of the huge number of Fortran variants out there, but by and large, GCC is the right solution to your compilation problems.

CHAPTER 1

Using GCC's C Compiler

This chapter's goal is to get you comfortable with typical usage of the GNU Compiler Collection's C compiler, gcc. This chapter focuses on those command-line options and constructs that are specific to GCC's C compiler. Options that can generally be used with any GCC compiler are discussed in Appendix A. Throughout this chapter, as throughout this book, I'll differentiate between GCC (the GNU Compiler Collection) and gcc, the C compiler that is provided as part of GCC.

This chapter explains how to tell gcc which dialect of C it should expect to encounter in your source files, from strict ANSI/ISO C to classic Kernighan and Ritchie (K&R) C. It also explains the variety of special-purpose constructs that are supported by gcc and how to invoke and use them. It concludes by discussing using gcc to compile Objective C applications and discusses specific details of the GNU Objective C runtime environment.

GCC Option Refresher

Appendix A discusses the options that are common to all of the GCC compilers and how to customize various portions of the compilation process. However, I'm not a big fan of making people jump around in a book for information. For that reason, this section provides a quick refresher of basic GCC compiler usage as it applies to the gcc C compiler. For detailed information, see Appendix A. If you are new to gcc and just want to get started quickly, you're in the right place.

The gcc compiler accepts both single-letter options, such as `-o`, and multiletter options, such as `-ansi`. Because it accepts both types of options you cannot group multiple single-letter options together as you may be used to doing in many GNU and Unix/Linux programs. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. The `-pg` option creates extra code in the final binary that outputs profile information for the GNU code profiler, gprof. On the other hand, the `-p -g` options generate extra code in the resulting binary that produces profiling information for use by the prof code profiler (`-p`) and causes gcc to generate debugging information using the operating system's normal format (`-g`).

Despite its sensitivity to the grouping of multiple single-letter options, you are generally free to mix the order of options and compiler arguments on the gcc command line. That is, invoking gcc as

```
gcc -pg -fno-strength-reduce -g myprog.c -o myprog
```

has the same result as

```
gcc myprog.c -o myprog -g -fno-strength-reduce -pg
```

I say that you are generally free to mix the order of options and compiler arguments because, in most cases, the order of options and their arguments does not matter. In some situations, order does matter if you use several options of the same kind. For example, the `-I` option specifies the directory or directories to search for include files. So if you specify `-I` several times, gcc searches the listed directories in the order specified.

Compiling a single source file, myprog.c, using gcc is easy—just invoke gcc, passing the name of the source file as the argument.

```
$ gcc myprog.c
$ ls -l
```

```
-rwxr-xr-x      1 wvh    users           13644 Oct 5 16:17 a.out
-rw-r--r--      1 wvh    users           220   Oct 5 16:17 myprog.c
```

By default, the result on Linux and Unix systems is an executable file named a.out in the current directory, which you execute by typing **./a.out**. On Cygwin systems, you will wind up with a file named a.exe that you can execute by typing either **./a** or **./a.exe**.

To define the name of the output file that gcc produces, use the `-o` option, as illustrated in the following example:

```
$ gcc myprog.c -o runme
$ ls -l
```

```
-rw-r--r--    1 wvh    users       220   Oct  5 16:17 myprog.c
-rwxr-xr-x    1 wvh    users       13644 Oct  5 16:28 runme
```

If you are compiling multiple source files using gcc, you can simply specify them all on the gcc command line, as in the following example, which leaves the compiled and linked executable in the file named showdate:

```
$ gcc showdate.c helper.c –o showdate
```

If you want to compile these files incrementally and eventually link them into a binary, you can use the `–c` option to halt compilation after producing an object file, as in the following example:

```
$ gcc -c showdate.c
$ gcc -c helper.c
$ gcc showdate.o helper.o –o showdate
$ ls -l
```

```
total 124
-rw-r--r--    1 wvh    users        210   Oct  5 12:42 helper.c
-rw-r--r--    1 wvh    users        45    Oct  5 12:29 helper.h
-rw-r--r--    1 wvh    users        1104  Oct  5 13:50 helper.o
-rwxr-xr-x    1 wvh    users        13891 Oct  5 13:51 showdate
-rw-r--r--    1 wvh    users        208   Oct  5 12:44 showdate.c
-rw-r--r--    1 wvh    users        1008  Oct  5 13:50 showdate.o
```

■ **Note** All of the GCC compilers "do the right thing" based on the extensions of the files provided on any GCC command line. Mapping file extensions to actions (for example, understanding that files with `.o` extensions only need to be linked) is done via the GCC specs file. Prior to GCC version 4, the specs file was a stand-alone text file that could be modified using a text editor; with GCC 4 and later, the specs file is built-in and must be extracted before it can be modified. For more information about working with the specs file, see the section "Customizing GCC Using Spec Strings" in Appendix A.

It should be easy to see that a project consisting of more than a few source code files would quickly become exceedingly tedious to compile from the command line, especially after you start adding search directories, optimizations, and other gcc options. The solution to this command-line tedium is the make utility, which is not discussed in this book due to space constraints (although it is touched upon in Chapter 8).

Compiling C Dialects

The gcc compiler supports a variety of dialects of C via a range of command-line options that enable both single features and ranges of features that are specific to particular variations of C. Why bother, you ask? The most common reason to compile code for a specific dialect of C is for portability. If you write code that might be compiled with several different tools, you can check for that code's adherence to a given standard using GCC support for various dialects and standards. Verifying adherence to various standards is one method developers use to reduce the risk of running into compile-time and runtime problems when code is moved from one platform to another, especially when the new platform was not considered when the program was originally written.

What then is wrong with vanilla ISO/ANSI C? Nothing that has not been corrected by officially ordained corrections. The original ANSI C standard, prosaically referred to as C89, is officially known as ANSI X3.159-1989. It was ratified by ANSI in 1989 and became an ISO standard, ISO/IEC9989:1990 to be precise, in 1990. Errors and slight modifications were made to C89 in technical corrigenda published in 1994 and 1996. A new standard, published in 1999, is known colloquially as C99 and officially as ISO/IEC9989:1999. The freshly minted C99 standard was amended by a corrigendum issued in 2001. This foray into the alphabet soup of standards explains why options are available for supporting multiple dialects of C. I'll explain how to use them a little later in this section.

In addition to the subtle variations that exist in standard C, some of the gcc C dialect options enable you to select the degree to which gcc complies with the standard. Other options enable you to select which C features you want. There is even a switch that enables limited support for traditional (pre-ISO, pre-ANSI) C. But enough discussion. Table 1-1 lists and briefly describes the options that control the C dialect to which gcc adheres during compilation.

Table 1-1. *C Dialect Command-Line Options*

Option	Description
`-ansi`	Supports all ISO C89 features and disables GNU extensions that conflict with the C89 standard.
`-aux-info` *file*	Saves prototypes and other identifying information about functions declared in a translation unit to the file identified by *file*.
`-fallow-single-precision`	Prevents promotion of single-precision operations to double-precision.

Table 1-1. *C Dialect Command-Line Options (Continued)*

Option	Description
`-fbuiltin`	Recognizes built-in functions that lack the `__builtin_` prefix.
`-fcond-mismatch`	Allows mismatched types in the second and third arguments of conditional statements.
`-ffreestanding`	Claims that compilation takes place in a freestanding (unhosted) environment. Freestanding means that the environment includes all of the library functions required to operate without loading or referencing external code. Currently, freestanding implementations provide all of the functions identified in `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`. Freestanding 64-bit code also requires the functions identified in `<iso646.h>`. Freestanding C99-compliant code also requires anything referenced in `<stdbool.h>` and `<stdint.h>`. The Linux kernel is a good example of a freestanding environment.
`-fhosted`	Claims that compilation takes place in a hosted environment, which means that external functions can be loaded from libraries such as the standard C library. This is the default value for GCC compilation. Programs that use external libraries (such as most applications) are good examples of applications that compile and execute in a hosted environment.
`-fno-asm`	Disables use of `asm`, `inline`, and `typeof` as keywords, allowing their use as identifiers.
`-fno-builtin`	Ignores built-in functions that lack the `__builtin_` prefix.
`-fno-signed-bitfields`	Indicates that bit fields of undeclared type are to be considered unsigned.
`-fno-signed-char`	Keeps the `char` type from being signed, as in the type `signed char`.
`-fno-unsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered signed.
`-fno-unsigned-char`	Keeps the `char` type from being unsigned, as in the type `unsigned char`.
`-fshort-wchar`	Forces the type `wchar_t` to be `short unsigned int`.
`-fsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered signed.
`-fsigned-char`	Permits the `char` type to be signed, as in the type `signed char`.
`-funsigned-bitfields`	Indicates that bit fields of undeclared type are to be considered unsigned.
`-funsigned-char`	Permits the `char` type to be unsigned, as in the type `unsigned char`.
`-fwritable-strings`	Permits strong constants to be written and stores them in the writable data segment.
`-no-integrated-cpp`	Invokes an external C preprocessor instead of the integrated preprocessor.
`-std=`*value*	Sets the language standard to value (`c89`, `iso9899:1990`, `iso9989:199409`, `c99`, `c9x`, `iso9899:1999`, `iso9989:199x`, `gnu89`, `gnu99`).

Table 1-1. *C Dialect Command-Line Options (Continued)*

Option	Description
`-traditional`	Supports a limited number of traditional (K&R) C constructs and features.
`-traditional-cpp`	Supports a limited number of traditional (K&R) C preprocessor constructs and features.
`-trigraphs`	Enables support for C89 trigraphs.

Sufficiently confused? Believe it or not, it breaks down more simply than it seems. To begin with, throw out `-aux-info` and `-trigraphs`, because you are unlikely to ever need them. Similarly, you are advised to not use `-no-integrated-cpp` because its semantics are subject to change and may, in fact, be removed from future versions of GCC. If you want to use an external preprocessor, use the `CPP` environment variable discussed in Appendix A or the corresponding `make` variable. Likewise, unless you are working with old code that assumes it can be scribbled into constant strings, do not use `-fwritable-strings`. After all, constant strings should be constant—if you are scribbling on them, they are variables, so just create a variable. To be fair, however, early C implementations allowed writable strings (primarily to limit stack space consumption), so this option exists to enable you to compile legacy code.

The various flags for signed or unsigned types exist to help you work with code that makes assumptions of the signedness of chars and bit fields. In the case of the char flags (`-fsigned-char`, `-funsigned-char`, and their corresponding negative forms), each machine has a default `char` type, which is either signed or unsigned. That is, given the statement

```
char c;
```

you might wind up with a `char` type that behaves like a `signed char` or an `unsigned char` on a given machine. If you pass gcc the `-fsigned-char` option, it will assume that all such unspecified declarations are equivalent to the statement

```
signed char c;
```

The converse applies if you pass gcc the `-funsigned-char` option. The purpose of these flags (and their negative forms) is to allow code that assumes the default machine `char` type is, say, like an unsigned char (that is, it performs operations on `char` types that assume an unsigned char), to work properly on a machine whose default `char` type is like a signed char. In this case, you would pass gcc the `-funsigned-char` option. A similar situation applies to the bit field–related options. In the case of bit fields, however, if the code does not specifically use the signed or unsigned keyword, gcc assumes the bit field is signed.

■ **Note** Truly portable code should not make such assumptions—that is, if you know you need a specific type of variable, say an unsigned char, you should declare it as such rather than using the generic type and making assumptions about its signedness that might be valid on one architecture but not on another.

You will rarely ever need to worry about the `-fhosted` and `-ffreestanding` options, but for completeness' sake, I'll explain what they mean and why they are important. In the world of C standards, an environment is either hosted or freestanding. A hosted environment refers to one in which the complete standard library is present and in which the program startup and termination occur via

a `main()` function that returns `int`. In a freestanding environment, on the other hand, the standard library may not exist and program startup and termination are implementation-defined. The implication of the difference is just this: in a freestanding implementation (when invoked with `-ffreestanding`), the gcc compiler makes very few assumptions about the meaning of function names that exist in the standard library. So, for example, the `ctime()` function is meaningless to gcc in freestanding mode. In hosted mode, which is the default, on the other hand, you can rely on the fact that the `ctime()` function behaves as defined in the C89 (or C99) standard.

■ **Note** This discussion simplifies the distinction between freestanding and hosted environments and ignores the distinction the ISO standard draws between conforming language implementations and program environments.

Now, about those options that control to which standards GCC adheres. Taking into account the command-line options I've already discussed, you are left with `-ansi`, `-std`, `-traditional`, `-traditional-cpp`, `-fno-asm`, `-fbuiltin`, and `-fno-builtin`. Here again, we can simplify matters somewhat. The `–traditional` option enables you to use features of pre-ISO C, and implies `-traditional-cpp`. These traditional C features include writable string constants (as with `-fwritable-strings`), the use of certain C89 keywords as identifiers (`inline`, `typeof`, `const`, `volatile`, and `signed`), and global extern declarations. You can see by looking at Table 1-1 that `-traditional` also implies `-fno-asm`, because `-fno-asm` disables the use of the `inline` and `typeof` keywords, such as `-traditional`, and also the `asm` keyword. In K&R C, these keywords could be used as identifiers.

The `-fno-builtin` flag disables recognition of built-in functions that do not begin with the `__builtin_` prefix. What exactly is a built-in function? Built-in functions are versions of functions in the standard C library that are implemented internally by gcc. Some built-ins are used internally by gcc, and only by gcc. These functions are subject to change, so they should not be used by non-GCC developers. Most of gcc's built-ins, though, are optimized versions of functions in the standard libraries, intended as faster and more efficient replacements of their externally defined cousins. You normally get these benefits for free because gcc uses its own versions of, say, `alloca()` or `memcpy()` instead of those defined in the standard C libraries. Invoking the `-fno-builtin` option disables this behavior. The GCC info pages document the complete list of gcc's built-in functions.

The `-ansi` and `-std` options, which force varying degrees of stricter adherence to published C standards documents, imply `-fno-builtin`. As Table 1-1 indicates, `-ansi` causes gcc to support all features of ISO C89 and turns off GNU extensions that conflict with this standard. To be clear, if you specify `-ansi`, you are selecting adherence to the C89 standard, not the C99 standard. The options `-std=c89` or `-std=iso9899:1990` have the same effect as `-ansi`. However, using any of these three options does not mean that gcc starts behaving as a strict ANSI compiler because GCC will not emit all of the diagnostic messages required by the standard. To obtain all of the diagnostic messages, you must also specify the options `-pedantic` or `-pedantic-errors`. If you want the diagnostics to be considered warnings, use `-pedantic`. If you want the diagnostics to be considered errors and thus to terminate compilation, use `-pedantic-errors`.

To select the C99 standard, use the option `-std=c99` or `-std-iso9989:1999`. Again, to see all of the diagnostic messages required by the C99 standard, use `-pedantic` or `-pedantic-errors` as previously described. To completely confuse things, the GNU folks provide arguments to the `-std` option that specify an intermediate level of standards compliance. Lacking explicit definition of a C dialect, gcc defaults to C89 mode with some additional GNU extensions to the C language. You can explicitly request this dialect by specifying `-std=gnu89`. If you want C99 mode with GNU extensions, you should specify, you guessed it, `-std=gnu99`. The default compiler dialect will change to `-std=gnu99` after gcc's C99 support has been completed.

What does turning on standards compliance do? Depending on whether you select C89 or C99 mode, the effects of `-ansi` or `-std=`*value* include